

# Dissertation

---

2025

---

Florian Deeg

---

***Automatenorientierte Entwurfsmethoden für  
Asynchrones Design in Programmierbaren  
Logikschaltungen***

---



# **Automatenorientierte Entwurfsmethoden für Asynchrones Design in Programmierbaren Logikschaltungen**

**Der Technischen Fakultät  
der Friedrich-Alexander-Universität  
Erlangen-Nürnberg**

**zur**

**Erlangung des Doktorgrades Dr. Ing.**

**vergelegt von**

**Florian Deeg**

**aus Reutlingen**

Als Dissertation genehmigt  
von der Technischen Fakultät  
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung: 18.03.2026

Gutachter/in: Prof. Dr. Sattler  
Prof. Dr. Fey

# Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mich auf dem Weg zu dieser Arbeit begleitet und unterstützt haben.

Mein besonderer Dank gilt meinem Betreuer Prof. Dr. Sebastian Michael Sattler, der mir nicht nur die nötigen Freiheiten gelassen hat, um kreativ zu werden, sondern gleichzeitig auch immer ein offenes Ohr für den wissenschaftlichen Diskurs hatte. Seine Unterstützung und sein Vertrauen haben diese Arbeit maßgeblich geprägt.

Ein ebenso großer Dank geht an meine Eltern, die mir überhaupt erst ermöglicht haben, mein Studium zu absolvieren. Ohne ihre Unterstützung, in jeglicher Hinsicht, wäre das alles nicht denkbar gewesen.

Den Mitarbeiterinnen und Mitarbeitern am Lehrstuhl möchte ich für die familiäre und angenehme Atmosphäre danken, die sie geschaffen haben. Ein besonderer Dank gilt meinen beiden Gym-Kollegen. Ich hoffe, dass sich unsere Wege nicht nur in der Wissenschaft, sondern auch weiterhin regelmäßig im Fitnessstudio kreuzen werden.

Meiner Trisch danke ich dafür, dass sie mich zwangsweise zu einem Frühaufsteher gemacht hat, auch wenn ich anfangs nicht unbedingt begeistert war, und immer beruhigen konnte, wenn es mal nicht so lief.

Zum Abschluss geht mein größter Dank an meine Freundin Katrin. Sie hat mich nicht nur mit klugen Gedanken im wissenschaftlichen Diskurs begleitet, sondern auch mit exzellenter Verköstigung versorgt, eine unschätzbare Kombination, die mich nicht nur mental, sondern auch körperlich durch diese intensive Zeit getragen hat.

# Kurzzusammenfassung

Diese Dissertation beschäftigt sich mit der Realisierung von asynchronen Automaten in Field Programmable Gate Arrays (FPGAs). Traditionelle synchrone Entwurfsmethoden stoßen in hochparallelisierten und energieeffizienten Systemen an ihre Grenzen, weshalb asynchrone Schaltungen eine vielversprechende Alternative darstellen.

Da FPGAs auf synchrones Design ausgelegt sind, wird zunächst die Umsetzbarkeit der asynchronen Entwurfsmethodiken mit besonderem Augenmerk auf den Freiheitsgrad im Entwurfsprozess analysiert. Hierfür werden verschiedene FPGAs und ihre Entwicklungsumgebungen verglichen und Begrenzungen der Entwurfsmöglichkeit aufgezeigt. Besonderer Fokus wird auf die Struktur der Look-Up-Table (LUT) gelegt, die als logische Komponente der Hauptbestandteil der asynchronen Schaltungen ist. Diese LUT wird dann verwendet, um ein funktionsstabiles Muller-C-Element zu realisieren.

Ein zentrales Problem in der asynchronen Schaltungstechnik ist die Vermeidung von kritischen Races und Hazards, die zu unerwartetem Systemverhalten führen können. Neben der formellen Betrachtung dieser Fehler, werden zur Lösung dieser Probleme verschiedene Kodierungstechniken untersucht, insbesondere basierend auf dem einschrittigen Automaten und dem Tree Subset Automaton (TSA). Die Analyse zeigt, dass die Einschritt-Kodierung schnell zu einem exponentiellen Anstieg der Zustandsdimensionen führt, wodurch der Schaltungsaufwand erheblich wächst. Durch geschickte Zerlegung in eine kaskadierte Struktur lässt sich jedoch eine Reduktion der benötigten Ressourcen erreichen. Diese kaskadierten Strukturen können dann als Dominologik aufgebaut werden.

Abschließend wird die Control Unit eines RISC-V Multi-Cycle-Prozessor als selbst-sperrende Dominologik-Schaltung realisiert. Zusätzlich wird auch eine Dominologik-Arithmetic Logic Unit (ALU) entworfen, um Handshaking im Prozessor zu ermöglichen. Der Prozessor ist weniger fehleranfällig, leistungsfähiger und energieeffizienter als sein synchrones Pendant bei leichtem Flächenanstieg.

# Abstract

This dissertation focuses on the implementation of asynchronous automata in the FPGA. Conventional synchronous design methodologies encounter limitations in highly parallelised and energy-efficient systems, thus highlighting the potential of asynchronous circuits as a promising alternative.

Given that FPGAs are designed for synchronous design, this study firstly analyses the feasibility of asynchronous design methodologies, with a particular emphasis on the degree of freedom in the design process. To this end, a comparative analysis of diverse FPGAs and their respective development environments is conducted, illuminating the constraints imposed on design possibilities. Particular emphasis is placed on the structure of the LUT, which serves as the primary logical component of asynchronous circuits. Following this, the LUT is employed to implement a functional stable Muller C-element.

A central problem in asynchronous circuit technology is the avoidance of critical races and hazards that can lead to unexpected system behaviour. In addition to the formal consideration of these errors, various coding techniques are investigated to solve these problems, in particular based on the one-step automaton and the TSA. The analysis shows that one-step coding quickly leads to an exponential increase in the state dimensions, which significantly increases the circuit complexity. However, a substantial reduction in the necessary resources can be accomplished through the skilful decomposition of the circuit into a cascaded structure, which can then be configured as domino logic.

Finally, the control unit of a RISC-V multi-cycle processor is implemented as a self-locking domino logic circuit, and a domino logic ALU is also designed to enable handshaking in the processor. The processor exhibits reduced error susceptibility, augmented processing capability, and enhanced energy efficiency in comparison to its synchronous counterpart, accompanied by a marginal increase in area.

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>I</b>
<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>IX</b>
<b>Abkürzungsverzeichnis</b>	<b>XI</b>
<b>Symbolverzeichnis</b>	<b>XIII</b>
<b>Glossar</b>	<b>XIII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Publikationen zu dieser Arbeit . . . . .	1
1.3 Ziel dieser Arbeit . . . . .	2
1.4 Aufbau der Dissertation . . . . .	2
<b>2 Grundlagen</b>	<b>4</b>
2.1 Logik . . . . .	4
2.1.1 Aussagen . . . . .	4
2.1.2 Aussagenlogik . . . . .	4
2.1.3 Positive Logik . . . . .	5
2.1.4 Grundlagen zur Booleschen Algebra . . . . .	6
2.1.5 Rechenregeln eines Booleschen Verbands . . . . .	8
2.1.6 Ableitungen Boolescher Funktionen . . . . .	10
2.1.7 Gerichtete Ableitungen . . . . .	12
2.1.8 Darstellung von binären Funktionen . . . . .	14
2.2 Ternäre Vektor Listen . . . . .	15
2.3 Kombinatorische Schaltungen . . . . .	16

---

2.4	Darstellung von Automaten . . . . .	17
2.4.1	Autonomer Automat . . . . .	18
2.4.2	Medwedew-Automat . . . . .	18
2.4.3	Moore-Automat . . . . .	18
2.4.4	Mealy-Automat . . . . .	19
2.4.5	Typ alt und Typ neu . . . . .	19
2.4.6	Vergleich der Automaten . . . . .	20
2.4.7	Z-Gleichungen und Automatengraphen . . . . .	20
2.4.8	Stabilisierungsarten eines Automaten . . . . .	21
2.5	Dynamische Effekte digitaler Schaltungen . . . . .	24
2.5.1	Hazards . . . . .	25
2.5.2	Races . . . . .	31
2.5.3	Glitches . . . . .	39
2.6	Asynchroner Entwurf . . . . .	41
2.6.1	Strukturtreue Modellierung . . . . .	42
2.6.2	Dual-Rail-Logik . . . . .	43
2.6.3	Muller-C-Element . . . . .	44
2.6.4	RS-Buffer . . . . .	45
2.6.5	Asynchrones stabiles Automaten-Design durch reduzierte Übertragungs-funktionen . . . . .	46
2.6.6	Handshake-Protokolle . . . . .	49
2.6.7	4-Phasen-Handshake-Protokoll . . . . .	50
2.6.8	2-Phasen-Handshake-Protokoll . . . . .	50
2.7	Einführung in die Krohn-Rhodes-Theorie . . . . .	51
2.8	Dominologik . . . . .	54
2.8.1	Motivation für Dominologik (DL) . . . . .	54
2.8.2	Funktionsweise eines DL-Gatters . . . . .	54
2.8.3	Kaskadierung von DL-Gattern . . . . .	55
2.8.4	Dual-Rail-Dominologik (DRDL)-Gatter . . . . .	57
<b>3</b>	<b>Asynchroner Entwurf in handelsüblichen FPGAs</b>	<b>60</b>

---

3.1	Einführung . . . . .	60
3.2	Field Programmable Gate Array . . . . .	61
3.2.1	Look-Up Tables . . . . .	64
3.2.2	Speicherelemente . . . . .	65
3.3	Der FPGA-Entwurfsfluss . . . . .	65
3.3.1	Hardware Description Languages . . . . .	67
3.3.2	Vergleich von VHDL und Verilog als Hardware Description Languages . . . . .	68
3.3.3	Logiksynthese . . . . .	68
3.3.4	Implementierungsprozess . . . . .	69
3.3.5	Bitstream-Generierung . . . . .	70
3.4	Gegenüberstellung der Frameworks . . . . .	70
3.4.1	Hintergrund . . . . .	70
3.4.2	Intel Altera . . . . .	71
3.4.3	Xilinx . . . . .	76
3.4.4	Fazit . . . . .	79
3.5	Realisierung eines funktionsstabilen Muller-C-Elements im Artix-7 . . . . .	80
3.5.1	Entwurf des funktionsstabilen Muller-C Elements . . . . .	81
3.5.2	Messung der Verzögerung des Rückkopplungspfads . . . . .	82
<b>4</b>	<b>Entwurfsmethoden Asynchroner Schaltungen</b>	<b>91</b>
4.1	Einschrittiges Automatendesign . . . . .	91
4.1.1	Einschrittige Kodierung der Eingangsvariablen . . . . .	91
4.1.2	Einschrittigkeit der z-Variablen . . . . .	93
4.1.3	Perfekter Automat . . . . .	95
4.1.4	Fazit zum taktlosen Automaten . . . . .	97
4.2	Selbstsperrende Dual-Rail-Dominologik . . . . .	98
4.2.1	Single-Rail-Dominologik im FPGA . . . . .	99
4.2.2	Dual-Rail-Dominologik . . . . .	100
4.2.3	Selbstsperrung . . . . .	101
4.2.4	Gesamte Dominologikschaltung . . . . .	105

---

4.2.5	Serielle Dominologik-Pipeline mit Vollständigkeitserkennung . . .	106
4.2.6	Parallelisierung von Dominogattern . . . . .	107
4.3	Vergleich der Entwurfsmethoden . . . . .	107
4.3.1	Einschrittiger Entwurf . . . . .	108
4.3.2	Kaskaden-Entwurf . . . . .	110
4.3.3	Vergleich der Entwürfe . . . . .	112
<b>5</b>	<b>Implementierung eines Asynchronen Multicycle Prozessors</b>	<b>115</b>
5.1	Motivation für RISC-V . . . . .	115
5.2	Globally Asynchronous Locally Synchronous (GALS) . . . . .	116
5.3	Realisierung des RISC-V-Prozessors . . . . .	116
5.3.1	Reduced Instruction Set Computer (RISC)-V Befehlssatz . . . .	117
5.3.2	Synchronous Multicycle CPU . . . . .	118
5.3.3	Synchrone Steuereinheit . . . . .	119
5.3.4	Entwurf des asynchronen Controllers . . . . .	119
5.3.5	Entwurf der asynchronen ALU . . . . .	121
5.3.6	Integration in die CPU . . . . .	123
5.4	Power, Performance and Area (PPA) Ergebnisse . . . . .	123
5.4.1	Leistungsanalyse . . . . .	124
5.4.2	Performanz-Analyse . . . . .	124
5.4.3	Flächenanalyse . . . . .	125
5.4.4	Diskussion . . . . .	126
5.5	Blueprint einer Asynchronen Multicycle CPU . . . . .	126
<b>6</b>	<b>Zusammenfassung und Ausblick</b>	<b>128</b>
6.1	Zusammenfassung . . . . .	128
6.2	Ausblick . . . . .	129
	Literaturverzeichnis . . . . .	131
A	Quellcode und Messdaten . . . . .	140

# Abbildungsverzeichnis

2.1	KV-Diagramm eines AA	5
2.2	KV-Diagramm eines Ausdrucks in PL	5
2.3	Priorisierung der Funktionen des BV	8
2.4	Graph der Funktionen $f = (x, 1 - x)$	12
2.5	Graph der binären Funktionen $f = (x, \bar{x})$	13
2.6	Beispiel eines KV-Diagramms	14
2.7	Signalflussplan des Booleschen Verbandsausdrucks	15
2.8	Allgemeine kombinatorische Schaltung	16
2.9	Autonomer Automat	18
2.10	Medwedjew-Automat	18
2.11	Moore-Automat	19
2.12	Mealy-Automat	19
2.13	Moore-Automat Typ alt und Typ neu	19
2.14	RS Buffer in TL	22
2.15	SFG eines Funktionsstabilisierten Automaten (zusätzlich ein funktionsstabilisierter isolierter Knoten)	23
2.16	Funktionsstabile Schaltung	24
2.17	Venn-Diagramm zur Einordnung von Funktions Hazards (F), Struktur Hazards (S) und Races (R)	25
2.18	Statischer Funktionshazard	25
2.19	Beispiel eines dynamischen Hazards	26
2.20	Struktur mit potentielltem Strukturhazard	29
2.21	Allgemeines Totzeitmodell	30
2.22	Das Low-Active RS-Latch	32
2.23	Veranschaulichung der Metastabilität	33
2.24	KV-Diagramme für $Q_1$ und $Q_0$	33
2.25	Untragbare Situation in der Race-Konstellation ( $Q_1 \sim Q_0$ )	34

---

2.26 Schaltplan des Automaten . . . . .	35
2.27 Race-Bedingung . . . . .	36
2.28 Race $z_1, z_0 = 11$ bzw. $00$ . . . . .	36
2.29 KV-Diagramme für $z_1$ und $z_0$ . . . . .	37
2.30 Allgemeine Asynchrone Feedback-Schaltung . . . . .	37
2.31 Venn-Diagramm der behandelten Asynchronen Schaltungen . . . . .	42
2.32 Umschaltvorgang bei der Dual-Rail-Logik . . . . .	43
2.33 Muller-C-Symbol . . . . .	44
2.34 Signalflussplan des Muller-C . . . . .	45
2.35 KV-Diagramm des RS-Buffers . . . . .	46
2.36 Vier Teile einer Zustandsüberföhrungsfunktion . . . . .	47
2.37 Gefaltetes KV-Diagramm . . . . .	47
2.38 Stabiler Moore-Automat . . . . .	49
2.39 4-Phasen-Handshake-Protokoll . . . . .	50
2.40 2-Phasen-Handshake-Protokoll . . . . .	51
2.41 Allgemeines Single-Rail-Dominologik (SRDL)-Gatter . . . . .	55
2.42 DL-Kaskade . . . . .	56
2.43 Impulsdigramm der Kaskade . . . . .	56
2.44 Dual-Rail-Dominologik auf TL . . . . .	58
2.45 Kaskade aus DRDL-Gattern . . . . .	59
3.1 FPGA-Struktur . . . . .	62
3.2 Struktur eines einfachen Logikblocks . . . . .	63
3.3 Zwischenverbindungen zwischen Logikblöcken im FPGA . . . . .	64
3.4 Zweistufiger Multiplexer (MUX) unter Verwendung von Passtransistoren . . . . .	65
3.5 FPGA-Flow . . . . .	66
3.6 Interconnect-Struktur des Cyclone II FPGA . . . . .	72
3.7 Muller C-element . . . . .	81
3.8 Struktur einer 6-Input LUT der Xilinx 7 Series FPGAs . . . . .	81
3.9 T-Buffer auf Transistorlevel (TL) und Gatterlevel (GL) . . . . .	83
3.10 N-stufiger Asynchroner Frequenzteiler . . . . .	83

3.11 Oszillierendes Signal für einen TB in LUT A des Slices X0Y0 unter Verwendung von Eingang A5 als Rückkopplungspfad. . . . .	85
3.12 Implementierung des TB in LUT A des Slices X0Y0 unter Verwendung von A5 oder A6 als Rückkopplungspfad (highlighted) . . . . .	86
3.13 FLPDs der TB implementiert in LUT A-D von Slice X0Y0-X3Y0 . . . . .	87
3.14 Rückkopplung auf eine sechs Eingänge umfassende LUT abgebildet . . . . .	88
3.15 Testschaltung zur Verifikation . . . . .	88
3.16 Testfälle . . . . .	89
3.17 Funktionsstabiles Muller-C-Element in einer sechs Eingängen umfassenden LUT realisiert . . . . .	90
4.1 Einschrittig x-kodierter Automat (Graph) . . . . .	92
4.2 KV-Diagramm von $\delta_{z_1}$ und $\delta_{z_0}$ . . . . .	93
4.3 Digitale Schaltung des Automaten . . . . .	93
4.4 Einschrittig kodierter Automat (Graph) . . . . .	94
4.5 Mehrschrittiger Automat (Graph) . . . . .	94
4.6 Richtungserkennung eines drehenden Objekts . . . . .	95
4.7 Einschrittig kodierter Rotationserkennungsautomat . . . . .	96
4.8 KV-Diagramm von $r = (R, \bar{R})$ . . . . .	96
4.9 KV-Diagramm von $z_1 = (Z_1, \bar{Z}_1)$ . . . . .	96
4.10 KV-Diagramm von $z_0 = (Z_0, \bar{Z}_0)$ . . . . .	97
4.11 Glitch-freie Dual-Rail-Implementierung . . . . .	97
4.12 Single-Rail Dominologik . . . . .	99
4.13 SRDL realisiert durch eine LUT3 . . . . .	99
4.14 Transitionen von SRDL abgebildet auf eine LUT3 . . . . .	100
4.15 Dual-Rail-Dominologik . . . . .	101
4.16 Pulsschaltung auf TL . . . . .	102
4.17 Pulsschaltung für Selbstsperrung und Duty Cycle . . . . .	103
4.18 SFG der Pulsschaltung . . . . .	103
4.19 Digitales Impulssdiagramm . . . . .	104
4.20 Selbstsperrende DRDL . . . . .	105
4.21 Beispielautomatengraph . . . . .	108

---

4.22	Umkodierter Beispielautomatengraph . . . . .	108
4.23	Partielles KV-Diagramm von $s_0$ und $\bar{s}_0$ . . . . .	109
4.24	Totales KV-Diagramm von $s_0 = (S_0, \bar{S}_0)$ . . . . .	110
4.25	Einschrittige Dual-Rail-Implementierung der Beispielschaltung . . . . .	111
4.26	Entwicklung der ersten Stufe des TSA . . . . .	111
4.27	Teil-TSA mit Inhibitionen und Zustandsübergangskanten . . . . .	111
4.28	TSA mit Inhibitionen und Zustandsübergangskanten . . . . .	112
4.29	Gatterlevel des TSA . . . . .	113
5.1	Synchroner CPU und die synchrone Control Unit . . . . .	118
5.2	Automatengraph (AG) des synchronen Moore-Automaten . . . . .	120
5.3	AG der Pipeline . . . . .	121
5.4	Struktur der realisierten Pipeline im FPGA . . . . .	121
5.5	ALU aus parallelen Dominogattern . . . . .	122
5.6	Leistungsverbrauch der Synchronen vs. Asynchronen CPU . . . . .	124
5.7	Asynchrone CPU . . . . .	126
A.1	Messung der Feedback-Delays . . . . .	155

# Tabellenverzeichnis

2.1	Zweistellige Boolesche Funktionen . . . . .	8
2.2	Beispielwahrheitstabelle . . . . .	14
2.3	Vergleich von Mealy- und Moore-Automaten . . . . .	20
2.4	Wertetabelle mit logischem und realem Zustand . . . . .	32
2.5	Wahrheitstabelle des Automaten . . . . .	35
2.6	Wahrheitstabelle des Muller-C . . . . .	44
2.7	Wahrheitstabelle des RS-Buffers . . . . .	45
2.8	Tabellarische Darstellung der vier Teile . . . . .	47
3.1	Phasenliste des T-Buffer . . . . .	82
3.2	Mittlere FLPD des TB in LUT A des Slices X0Y0 . . . . .	85
3.3	Minimum FLPD der LUTs A-D der Slices X0Y0-X3Y0 . . . . .	87
4.1	Phasenliste des Automaten . . . . .	92
4.2	Wahrheitstabelle der Pulsschaltung . . . . .	102
4.3	Zustandsüberführungstabelle des partiellen Race-freien Automaten . . . . .	109
5.1	RISC-V Instruktionsformate . . . . .	117
5.2	Performanz-Kennzahlen . . . . .	125
5.3	FPGA Ressourcenverwendung . . . . .	125



# Abkürzungsverzeichnis

<b>BA</b>	Boolesche Algebra
<b>BV</b>	Boolescher Verband
<b>KVD</b>	Karnaugh-Veitch-Diagramm
<b>AA</b>	Aussagenlogischer Ausdruck
<b>AA<sub>s</sub></b>	Aussagenlogische Ausdrücke
<b>PL</b>	Positive Logik
<b>KBS</b>	Kleinbuchstaben
<b>GBS</b>	Großbuchstabe
<b>ZÜF</b>	Zustandsüberföhrungsfunktion
<b>TV</b>	Ternärvektor
<b>TVL</b>	Ternärvektorlisten
<b>RSB</b>	RS-Buffer
<b>DNF</b>	Disjunktive Normalform
<b>SEU</b>	Single Event Upset
<b>MUX</b>	Multiplexer
<b>TL</b>	Transistorlevel
<b>GL</b>	Gatterlevel
<b>PU</b>	Pull-Up
<b>PD</b>	Pull-Down
<b>DL</b>	Dominologik
<b>SRDL</b>	Single-Rail-Dominologik
<b>DRDL</b>	Dual-Rail-Dominologik

---

<b>LUT</b>	Look-Up-Table
<b>FPGA</b>	Field Programmable Gate Array
<b>CLB</b>	Configurable Logic Block
<b>LAB</b>	Logic Array Block
<b>LE</b>	Logic Element
<b>SoC</b>	System on Chip
<b>DFF</b>	D-Flipflop
<b>GALS</b>	Globally Asynchronous Locally Synchronous
<b>PIP</b>	Programmable Interconnect Points
<b>RTL</b>	Register-Transfer-Level
<b>HDL</b>	Hardware Description Language
<b>VHDL</b>	Very High-speed Integrated Circuit Hardware Description Language
<b>ASIC</b>	Anwendungsspezifische integrierte Schaltung
<b>VDS</b>	Vivado Design Suite
<b>FLPD</b>	Feedback Loop Path Delay
<b>TB</b>	T-Buffer
<b>AG</b>	Automatengraph
<b>TSA</b>	Tree Subset Automaton
<b>RISC</b>	Reduced Instruction Set Computer
<b>MS</b>	Multi-Set
<b>ALU</b>	Arithmetic Logic Unit
<b>PPA</b>	Power, Performance and Area
<b>ISA</b>	Instruction Set Architecture
<b>SPEC</b>	Standard Performance Evaluation Corporation

# Symbolverzeichnis

*	nicht definiert.
$\vee$	Supremum, Logische Oder-Verknüpfung
$\wedge$	Infimum, Logische Und-Verknüpfung
$\neg$	Logische Negation
$\delta$	Zustandsüberföhrungsfunktion
$\Sigma$	Eingabealphabet
$Z$	Zustandsmenge
$x$	Eingangvariable
$Z_1$	Zustand 1
$z$	z-Variable

# Glossar

## **Asynchrone Schaltung**

Eine Schaltung, die ohne einen globalen Takt arbeitet, also alles außer synchron.

## **Control Unit**

Steuereinheit im Sinne eines Automaten.

## **Handshaking-Protokoll**

Ein Protokoll, was zwischen einem Sender und Empfänger die ereignisgesteuerte Datenübertragung gewährleistet.

## **Hazard**

Ein unerwartetes Verhalten in einer digitalen Schaltung, das durch Verzögerungen passiert.

## **Late- und Early-arrival Signale**

Die Eingangssignale, die später als die übrigen Signale an einem Gatter oder einer Schaltung eintreffen, und die Eingangssignale, die früher als andere Signale ankommen.

## **Kaskadenstruktur**

Eine kaskadierte Schaltung mit late- und early-arrival Signalen.

## **Lookup Table**

Eine Struktur zur Implementierung logischer Funktionen in FPGA-Designs.

## **Low-Level-Entwurf**

Entwurf auf Transistor-Level.

## **Race**

Ein Wettlauf von Signalen auf den Rückkoppelleitungen in einer digitalen Schaltung.

## **Self-resetting**

Sich selbst zurücksetzende Schaltung.

## **(Quasi-)Delay-Insensitive**

Entwurfstil für asynchrone Schaltungen, der (in definierten Grenzen) unabhängig von Signallaufzeiten ist.

**Globally Asynchronous Locally Synchronous**

Architekturprinzip, bei dem mehrere synchrone Teilsysteme über asynchrone Schnittstellen verbunden werden.

**Pipeline**

Architekturprinzip, bei dem Verarbeitungsschritte in Teiletappen zerlegt und parallelisiert werden.

# 1 Einleitung

## 1.1 Motivation

Obwohl synchrone Schaltungen die fortgeschrittenere und weitverbreitete Technologie darstellen, hat die Berücksichtigung asynchroner Schaltungen mit ihren verschiedenen Vorteilen (geringerer Stromverbrauch, bessere Systemleistung und keine Probleme mit Taktverzögerungen) in den letzten Jahren zunehmend an Bedeutung gewonnen [13]. Synchrone Schaltungen passen ihren Takt an die ungünstigste Pfadverzögerung an, um dem System genügend Zeit zur Verarbeitung zu geben, was die Geschwindigkeit der Schaltung deutlich einschränkt. Das Taktsignal stellt zudem noch einen Single-Point-of-Failure in der Schaltung dar. Um eine höhere Systemleistung und System-sicherheit zu erreichen, besteht das offensichtliche Ziel darin, den globalen Taktgeber durch den Entwurf asynchroner Schaltungen zu vermeiden. Es ist jedoch wichtig, die Verzögerungszeiten der verwendeten Schaltungen zu kennen und zu kompensieren, um stabile Schaltungen zu erhalten, da andernfalls Fehler wie z. B. Races auftreten können. Field Programmable Gate Arrays (FPGA) sind eine gute Wahl für das Prototyping von Schaltungen, da es sich um kommerziell erhältliche Bausteine handelt. Diese Standardbausteine haben sich aufgrund ihrer Programmierbarkeit, ihrer Verfügbarkeit und der zugehörigen EDA-Tools (Electronic Design Automation) als äußerst nützlich für die Verifikation von anwendungsspezifischen integrierten Schaltungen (ASICs) erwiesen. Darüber hinaus haben sich FPGAs von einem Prototypen zu einer endgültigen Realisierung von Hardware entwickelt, die als Accelerator bezeichnet wird, und können neue Möglichkeiten im Automobilsektor eröffnen, indem sie die Rekonfiguration von Hardware ermöglichen und im Vergleich zu Softwarelösungen eine schnellere Hardwareverarbeitung, z. B. durch Parallelität, erlauben [4]. Um sie jedoch für asynchrone Schaltungen nutzen zu können, müssen Standardprozeduren angepasst werden.

## 1.2 Publikationen zu dieser Arbeit

Die in dieser Arbeit präsentierten Ergebnisse sind teilweise in den folgenden Veröffentlichungen enthalten:

- Unterabschnitt 2.4.8 und Abschnitt 3.5 basieren auf [1]. Der Hardwarevergleich,

der Testentwurf sowie die meisten Teile der vorliegenden Arbeit wurden vom Autor dieser Thesis verfasst.

- Unterabschnitt 3.2.1 und Abschnitt 2.5 basiert auf [2]. Die theoretischen Grundlagen, die Beispiele und Implementierung sowie die Messungen und die meisten Teile der Arbeit wurden vom Autor dieser Thesis verfasst.
- Abschnitt 3.5 basiert auf [1]. Die theoretischen Grundlagen, die Stabilisierungsarten und der Testentwurf stammen vom Autor dieser Arbeit. Zudem hat er die meisten Teile des Papers verfasst.
- Abschnitt 4.1 basiert auf [3]. Die meisten Teile dieser Arbeit wurden vom Autor dieser Thesis verfasst, dazu zählen die theoretischen Grundlagen und die Beispiele zur Kodierung sowie der Beispielautomat zur Realisierung des Rotationsdetektors.
- Abschnitt 3.1 bis Abschnitt 3.4 basieren teilweise auf [4]–[6], die vom Autor dieser Arbeit betreut wurden.
- Abschnitt 4.2 und Kapitel 5 basiert auf den Ergebnissen präsentiert in [7]–[9]. Die theoretischen Grundlagen, die Beispiele und Implementierung sowie die Messungen und die meisten Teile dieser Arbeit wurden vom Autor dieser Thesis verfasst.

### **1.3 Ziel dieser Arbeit**

Ziel der Arbeit ist es, asynchrone Design-Methoden im FPGA zu verifizieren und auf ihre Vor- und Nachteile zu analysieren. Dieses Ziel stellt eine große Herausforderung dar, da FPGAs im Allgemeinen für synchrones Design ausgelegt sind und es hierfür Adaptionen im Designprozess bedarf. Es sollen zunächst einige asynchrone Entwurfsmethoden dargestellt, mit besonderem Hinblick auf den FPGA-Entwurf auf ihre Vor- und Nachteile untersucht und abschließend ein asynchroner Mikroprozessor realisiert werden. Auf die Verifikation der Sicherheit des Entwurfs (strukturtreue Modellierung) wird dabei ein besonderes Augenmerk geworfen.

### **1.4 Aufbau der Dissertation**

Die Dissertation ist in sechs Kapitel unterteilt. Beginnend mit der Einleitung in Kapitel 1, in der die Problemstellung und Zielsetzung der Arbeit vorgestellt wird.

---

In Kapitel 2 werden die theoretische Grundlagen, die für das Verständnis der Arbeit vorausgesetzt werden, näher beschrieben.

Kapitel 3 beschäftigt sich mit verschiedenen FPGAs und deren Entwicklungsumgebung und zeigt, wie der Entwurfsprozess angepasst werden muss, um asynchron entwerfen zu können. Abschließend wird das Muller-C-Element als grundlegendes asynchrones Bauteil im FPGA funktionsstabil entworfen.

Anschließend werden verschiedene asynchrone Entwurfsmethoden und ihre Realisierung im FPGA in Kapitel 4 thematisiert.

Kapitel 5 zeigt den Entwurf und die Vorteile einer asynchronen Control-Unit und ALU für einen RISC-V Prozessor.

Der letzte Abschnitt Kapitel 6 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick ab.

## 2 Grundlagen

Die für diese Arbeit wichtigen Grundlagen werden in diesem Kapitel kurz erläutert. Die Grundlagen dienen dem Verständnis der asynchronen Entwurfsprozesse, der Probleme, die mit einer fehlenden globalen Synchronisation einhergehen sowie deren Behebungen aus schaltalgebraischer Sicht.

### 2.1 Logik

Hauptwerkzeug zur Beschreibung von digitalen Systemen stellt die Logik ausgehend von der Wertigkeit (Körper, =) dar, die sich als Schalter geöffnet oder ungeöffnet betrachten lässt, anders ausgedrückt Spannung anliegend oder frei von Spannung. Die folgenden Definitionen sind mit wenigen Änderungen bzw. Zusätzen aus [10], [11] und [12] übernommen.

#### 2.1.1 Aussagen

Aussagen bilden den, durch den Menschen erfassten, Ursprung für digitale Systeme. So wird, z.B. mittels eines Lastenhefts, durch Aussagen zunächst erfasst, was gewünscht wird. In diesem Fall ist z.B. "Der Automat leuchtet rot" eine Aussage. Der Automat, also das Subjekt, hat also die Eigenschaft, dass er rot leuchtet. Diese Aussage muss wahr sein und liegt als  $w$  oder  $\bar{f}$  vor. So wird die folgende Symbolik in [11] definiert:

##### **Definition 1** Aussagen

1. Aussagenvariable:  $A, B, \dots$  sind Symbole für Aussagen in Großbuchstabe (GBS).
2. Elementare Aussagen beschreiben eine Eigenschaft (Prädikat)  $p, q, \dots$  eines Individuums  $x_0, x_1, \dots$  aus einem bestimmten Individuenbereich  $\{x_0, x_1, \dots\}$ .

#### 2.1.2 Aussagenlogik

In dieser Dissertation wird die Aussagenlogik als Logik unärer Variablen behandelt, d.h. ein Aussagenlogischer Ausdruck (AA) hat genau einen Wert und liegt nicht als binäre

Variable vor. Da Aussagen immer wahr sind, handelt es sich bei AAs um  $w$ -Ausdrücke, die unär als  $w$  oder  $\bar{f}$  vorliegen ( $\bar{f}$  ist ein  $w$ -Ausdruck). Das Beispiel

$$\text{AA: } \bar{Y} = X_0\bar{X}_1 \vee \bar{X}_0X_1 \vee \bar{X}_2\bar{X}_1, \quad \langle \bar{Y} \rangle = X^1$$

ist in Abbildung 2.1 im KV-Diagramm dargestellt. Man kann diese  $w$ -Ausdrücke unter einem 1-Symbol subsummieren. Im Folgenden werden Aussagenlogische Aus-

	— $x_0$ —			
$\bar{Y}$	0	1	3	2
	$\bar{f}$	$\bar{f}$	*	$\bar{f}$
$x_2$	4	5	7	6
	*	$\bar{f}$	*	$\bar{f}$
	— $x_1$ —			

Abbildung 2.1: KV-Diagramm eines AA

drücke (AAs) als GBS dargestellt, um anzudeuten, dass es sich um eine unäre Variable handelt und somit eine partielle Funktion vorliegt.

### 2.1.3 Positive Logik

Analog zur AA sind Formeln in Positiver Logik (PL) 1-Ausdrücke, die unär als 1 oder 0 vorliegen. Die 0 ist dabei eine 1. Die 1 oder 0 verstehen sich hier jedoch als Symbol und nicht als Wert. Nur die Boolesche Algebra (BA) hat den Wert 0 und 1. Das Beispiel

$$\text{PL: } (\bar{Y}, Y) = (X_2X_0, \bar{X}_2\bar{X}_1), \quad \langle \bar{Y} \rangle = X^0, \quad \langle Y \rangle = X^1$$

ist in Abbildung 2.2 im KV-Diagramm für  $y = (Y, \bar{Y})$  dargestellt. Man kann diese Sym-

	— $X_0$ —			
$y$	0	1	3	2
	1	1	*	*
$X_2$	4	5	7	6
	*	0	0	*
	— $X_1$ —			

Abbildung 2.2: KV-Diagramm eines Ausdrucks in PL

bole auch unter einem 1-Symbol subsummieren. PL Ausdrücke werden in GBS dargestellt und sind eine geeignete Beschreibung für den TL. Die PL ist eine partielle Logik.

### 2.1.4 Grundlagen zur Booleschen Algebra

Eine konkrete Algebra besteht aus einer Menge von Wertebereichen, einer Menge von Funktionen sowie einer Menge von Operationen. So kann eine konkrete Algebra allgemein definiert werden [13]:

**Definition 2** *Konkrete Algebra  $A_k$*

mit:

- Menge von Wertebereichen  $W_k$
- Menge von Funktionen  $F_k$
- Menge von Operationen  $Q$

So kann die BA definiert werden:

**Definition 3** *Boolesche Algebra (BA)*

mit:

- Menge von Wertebereichen  $W_k = \{0, 1\}$
- Menge von Operationen  $Q = \{ \cdot, +, \bar{\cdot} \}$
- Menge von Funktionen  $F_k = \{2^{2^n}\}$

Die Algebraische Struktur besteht also aus einer Trägermenge von Verknüpfungen und der Menge aller in ihrem Wertverlauf unterschiedlichen Funktionen. Multisets sollen durch ihren kleinsten Repräsentanten dargestellt werden. Kombinatorische und sequentielle binäre Systeme können durch Funktionen der Booleschen Algebra dargestellt werden. Eine Boolesche Funktion ordnet jede Eingangskombination  $x$  genau einer Ausgangskombination  $y$  zu. Somit ergibt sich für die  $m$ -stellige Boolesche Funktion  $\varphi$  mit einem Ausgang:

$$y = \varphi(x_{m-1}, \dots, x_1, x_0) \tag{2.1}$$

Eine digitale Schaltung kann zudem durch die Boolesche Mengenalgebra konkretisiert werden [11], indem  $m$  Ausgänge durch eine Menge von  $m$  Booleschen Funktionen der Form

$$\{0, 1\}^n \rightarrow \{0, 1\} \quad (2.2)$$

dargestellt werden. Jeder Ausgang besitzt also eine konkretisierte Funktion und jeder Eingang führt zu genau einem Ausgang. Die Menge der verschiedenen Funktionen bei  $n$  Eingängen lässt sich ermitteln mit  $2^{2^n}$ . Die Boolesche Mengenalgebra kann nun definiert werden [11], [14]:

**Definition 4** *Boolesche Mengenalgebra (BMA)*

BMA =  $[P(X), \cup, \cap, \bar{\cdot}, \emptyset, X]$  mit:

- $P(X)$  als Trägermenge
- $\cup, \cap, \bar{\cdot}$  als Operationen
- $\emptyset$  als neutrales Element der Vereinigung ( $\cup$ )
- $X$  als neutrales Element der Schnittbildung ( $\cap$ )

isomorph ist auch die Boolesche Ausdrucksalgebra der Booleschen Mengenalgebra. Sie dient der Beschreibung von Booleschen Funktionen und bildet Terme, die direkt als Schaltungselemente gesehen werden können. Die Boolesche Ausdrucksalgebra bzw. Schaltalgebra wird definiert [11] als:

**Definition 5** *Boolesche Ausdrucksalgebra (BAA)*

BAA =  $[H, \vee, \wedge, \bar{\cdot}, 0, 1]$

mit

- $H$  als Trägermenge
- $\vee, \wedge, \bar{\cdot}$  als Operationen
- $0$  als neutrales Element der Disjunktion
- $1$  als neutrales Element der Konjunktion

Im Folgenden sollen die zweistelligen Booleschen Grundfunktionen kurz in Tabelle 2.1 aufgeführt werden [10]. Da die zwei Variablen  $x_1$  und  $x_0$  genau  $2^2$  verschiedene Kombi-

nationen aufweisen, entstehen  $2^{2^2}$  also 16 verschiedene Ausgangskombinationen, also die Trägermenge des Booleschen Verbands.

$x_1$	0	0	1	1	Aussage	Symbol	Bezeichnung
$x_0$	0	1	0	1			
$y_0$	0	0	0	0	konstant 0	0	
$y_1$	0	0	0	1	$x_1$ und $x_0$	$x_1 \wedge x_0$	Konjunktion
$y_2$	0	0	1	0	$x_1$ und nicht $x_0$	$x_1 \wedge \bar{x}_0$	Inhibition
$y_3$	0	0	1	1	gleich $x_1$	$x_1$	Identität
$y_4$	0	1	0	0	nicht $x_1$ und $x_0$	$\bar{x}_1 \wedge x_0$	Inhibition
$y_5$	0	1	0	1	gleich $x_0$	$x_0$	Identität
$y_6$	0	1	1	0	$x_1$ antivalent $x_0$	$x_1 \approx x_0$	Antivalenz
$y_7$	0	1	1	1	$x_1$ oder $x_0$	$x_1 \vee x_0$	Disjunktion
$y_8$	1	0	0	0	weder $x_1$ noch $x_0$	$\overline{x_1 \vee x_0}$	NOR
$y_9$	1	0	0	1	$x_1$ äquivalent $x_0$	$x_1 \sim x_0$	Äquivalenz
$y_{10}$	1	0	1	0	nicht $x_0$	$\bar{x}_0$	Negation
$y_{11}$	1	0	1	1	$x_0$ impliziert $x_1$	$x_0 \rightarrow x_1$	Implikation
$y_{12}$	1	1	0	0	nicht $x_1$	$\bar{x}_1$	Negation
$y_{13}$	1	1	0	1	$x_1$ impliziert $x_0$	$x_1 \rightarrow x_0$	Implikation
$y_{14}$	1	1	1	0	nicht ( $x_1$ und $x_0$ )	$\overline{x_1 \wedge x_0}$	NAND
$y_{15}$	1	1	1	1	konstant 1	1	

Tabelle 2.1: Zweistellige Boolesche Funktionen

### 2.1.5 Rechenregeln eines Booleschen Verbands

Die für diese Thesis festgelegte Reihenfolge der Priorisierung der Operationen ist in Abbildung 2.3 dargestellt [10].

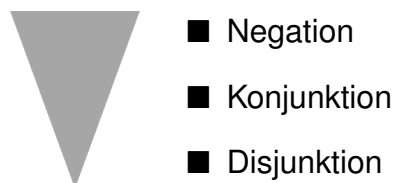


Abbildung 2.3: Priorisierung der Funktionen des BV

Im Sinne dieser Auslegung wird in dieser Arbeit die Negation von  $x$  geschrieben als  $\bar{x}$ , sowie die Konjunktion vereinfacht:  $x_1 \wedge x_0 = x_1 x_0$ . Der Boolescher Verband (BV) beinhaltet außerdem Gesetze. Hierzu zählen [11]:

**Definition 6** *Gesetze des BV*

- Das Kommutativgesetz:  $x_1x_0 = x_0x_1$
- Das Assoziativgesetz:  $x_2 \wedge (x_1x_0) = (x_2x_1) \wedge x_0 = x_2x_1x_0$
- Das Distributivgesetz:  $x_2 \wedge (x_1 \vee x_0) = x_2x_1 \vee x_2x_0$
- Das Idempotenzgesetz:  $x \wedge x = x$
- Das Absorptionsgesetz:  $x_1 \wedge (x_1 \vee x_0) = x_1$

Weitere Rechenregeln sind in den Gleichungen 2.3 bis 2.8 gegeben.

$$x \wedge \bar{x} = 0 \quad (2.3)$$

$$x \vee \bar{x} = 1 \quad (2.4)$$

$$x \wedge 1 = x \quad (2.5)$$

$$x \wedge 0 = 0 \quad (2.6)$$

$$x \vee 1 = 1 \quad (2.7)$$

$$x \vee 0 = x \quad (2.8)$$

$$(2.9)$$

Die DeMorgan'schen Gesetze werden in den Gleichungen 2.10 und 2.11 gezeigt.

$$\overline{(x_1 \wedge x_0)} = \bar{x}_1 \vee \bar{x}_0 \quad (2.10)$$

$$\overline{(x_1 \vee x_0)} = \bar{x}_1 \wedge \bar{x}_0 \quad (2.11)$$

Zudem gilt die Negation der Negation in Gleichung 2.12 .

$$\overline{\bar{x}} = x \quad (2.12)$$

Diese Regeln können nun zum Kürzen von Termen verwendet werden, indem z.B. mittels Distributivgesetz die Terme geschickt kombiniert werden, siehe Gleichung 2.13.

$$x_1x_0 \vee x_1\bar{x}_0 = x_1 \wedge (x_0 \vee \bar{x}_0) = x_1 \wedge 1 = x_1 \quad (2.13)$$

Der BV lässt sich definieren durch:

**Definition 7** *Boolescher Verband (BV)*

$$BV = [\wedge, \vee, \bar{\phantom{x}}, 0, 1]$$

mit

- $\wedge, \vee, \bar{\phantom{x}}$  als Operationen
- 0 als neutrales Element der Disjunktion
- 1 als neutrales Element der Konjunktion

### 2.1.6 Ableitungen Boolescher Funktionen

Ableitungen Boolescher Funktionen geben an, ob sich der Wert der Funktion ändert, wenn sich die Variable, nach der abgeleitet wird, ändert. Formal zeigt die Ableitung den Teil der Funktion, der von der Variablen abhängig ist, also je nach Zustand der Variablen einen anderen Ausgangswert einnimmt. Das gilt natürlich auch für Ableitungen nach mehreren Variablen. Die einzelnen Varianten der Ableitungen Boolescher Funktionen [10] sollen im Folgenden kurz erläutert werden.

#### Einfache partielle Ableitung

Um herauszufinden, ob eine gegebene Funktion  $f$  von einer Variablen abhängig ist, kann die einfache partielle Ableitung angewandt werden. Die allgemeine Formel für die einfache partielle Ableitung der Booleschen Funktion  $f$  nach  $x_i$  lautet:

$$\frac{df}{dx_i} = f(x_i) \approx f(\bar{x}_i) \quad (2.14)$$

#### Mehrfache partielle Ableitung

Partielle Ableitungen können auch mehrfach nacheinander abgeleitet werden. So ergibt sich die Formel für die k-fache partielle Ableitung:

$$\frac{d^k f}{dx_{k-1} dx_{k-2} \dots dx_1 dx_0} = \frac{d}{dx_{k-1}} \left( \frac{d}{dx_{k-2}} \left( \dots \frac{d}{dx_1} \left( \frac{df}{dx_0} \right) \dots \right) \right) \quad (2.15)$$

## Vektorielle Ableitung

Die vektorielle Ableitung zeigt an, ob sich der Wert einer Funktion ändert, wenn sich die Eingänge gleichzeitig ändern und wird berechnet, indem der Eingangsvektor  $x$  in zwei Teilvektoren  $x_p$  und  $x_q$  zerlegt wird, durch:

$$\left. \frac{df(x)}{dx_p} \right|_{x_p} = f(x_p, x_q) \approx f(x_p \approx 1, x_q) \text{ mit } x = (x_p, x_q) \quad (2.16)$$

## Variationsableitung

Die Variationsableitung untersucht, ob sich der Funktionswert einer Booleschen Funktion  $f$  ändert, wenn sich, ausgehend von einer Anfangsbelegung  $\underline{x}^a$ , beliebige Komponenten des Variablenvektors  $\underline{x}$  ändern. Formal ausgedrückt bedeutet das:

$$\left. \frac{\Delta f(x)}{\Delta x_p} \right|_{x_p} = \bigvee_c f(x_p, x_q) \approx f(x_p \approx c, x_q) \quad (2.17)$$

mit  $x_p = (x_1, x_0)$  und  $\underline{0} < c \leq \underline{1}$

$\underline{0}$  steht dabei für die Belegung  $X_0$  von  $x_p$ , wobei jede Stelle mit der 0 belegt ist, und  $\underline{1}$  Belegung  $X_{N-1}$  von  $x_p$ , wobei jede Stelle mit 1 belegt ist.  $N$  ist dabei die Anzahl der Dimensionen des Vektors  $x_p$ .

## Differential

Wird für die vektorielle Ableitung bei der Antivalenz anstelle des Wertes 1 ein variabler Vektor  $dx_p$ , von dem beliebige Komponenten den Wert 1 annehmen können, eingefügt, so erhält man die vektorielle Ableitung in parametrischer Form, welche als Differential bezeichnet wird.

$$df(x_p, x_q) = f(x_p, x_q) \approx f(x_p \approx dx_p, x_q) \quad (2.18)$$

## Variation

Auch die Variationsableitung kann in parametrischer Form, genannt Variation, dargestellt werden:

$$\delta f(x_p, x_q) = \bigvee_c (f(x_p, x_q) \approx f(x_p \approx c dx_p, x_q)) \quad (2.19)$$

mit  $0 < c \leq 1$

### 2.1.7 Gerichtete Ableitungen

Im vorherigen Abschnitt wurde erläutert, wie z.B. einfach partielle Ableitungen berechnet werden. Diese sagen aus, für welchen Bereich, also für welche Vorbedingungen (testmodes) die Funktion abhängig von der Eingangsvariablen ist. Verloren geht bei dieser Betrachtung jedoch, ob es sich um eine positive oder negative Abhängigkeit handelt, d.h. gibt es eine positive oder negative Transition für einen Eingangswechsel von 1 nach 0. Um zu erläutern, wie diese Ableitungen genauer betrachtet werden, sodass eine Richtung zu erkennen ist, wird zunächst einmal die Berechnung einer linearen Funktion in der Körperalgebra betrachtet.

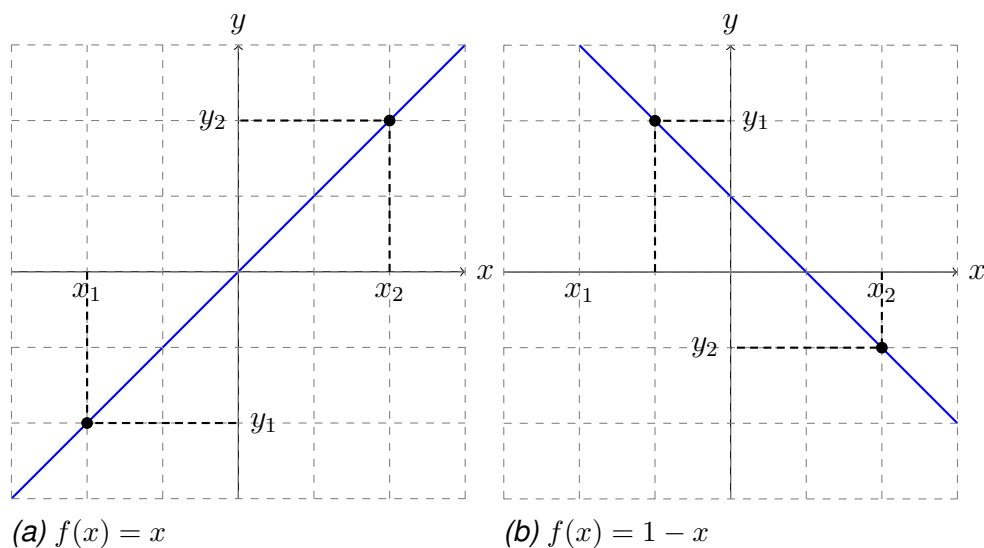


Abbildung 2.4: Graph der Funktionen  $f = (x, 1 - x)$

In Abbildung 2.4(a) ist die lineare Funktion  $f(x) = x$  dargestellt. Um die Steigung  $m$  zu berechnen, werden nun zwei Funktionswerte  $f(x_2)$  und  $f(x_1)$  an den Stellen  $x_2$  und

$x_1$  betrachtet und von  $x_2$  nach  $x_1$  voneinander abgezogen zu  $\frac{f(x_2)-f(x_1)}{x_2-x_1}$ . Es wird also verglichen, wie sich der Funktionswert von  $x_1$  nach  $x_2$  ändert. Gleiches gilt für die Funktion  $f(x) = 1 - x$  in Abbildung 2.4(b), welche in der Booleschen Algebra der Komplementbildung entspricht, da für  $x = 1$  das Ergebnis eine 0 ist und für  $x = 0$  eine 1 als Ergebnis vorliegt. Würde die einfach partielle Ableitung einfach angeben, dass die Funktion abhängig vom Eingang  $x$  ist, gibt die Ableitung  $m$  noch zusätzlich die Richtung mit dem Vorzeichen an, also im Fall (a)  $m = 1$  und Fall (b)  $m = -1$ .

Ziel ist es nun, dieses Wissen auf ein binäres System zu übertragen. Im binären System beschränken wir uns auf die 1 und 0, d.h.  $f(x) = x$  beschreibt nun die gleiche Funktion wie oben nur in einem binären System. Auf der  $x$ -Achse wird nun  $x_2$  mit 1 und  $x_1$  mit 0 belegt, wir landen beim Grafen aus Abbildung 2.5(a). In Abbildung 2.5(b) wird  $x$  gleich belegt und  $f(x) = 1 - x$  nimmt das Komplement von  $x$  an, es kann also auch geschrieben werden als  $f(x) = \bar{x}$ . Spannt man einen Bereich von  $[00]$  nach

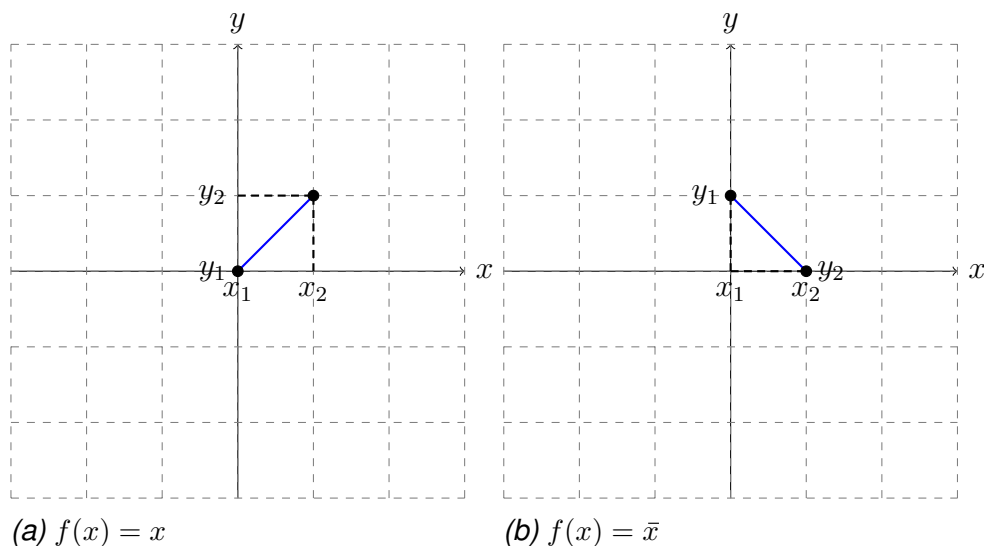


Abbildung 2.5: Graph der binären Funktionen  $f = (x, \bar{x})$

[11] auf, liegt der gleiche Graph vor. Im Binären könnte von einer Transition gesprochen werden. Wird nun der Anfangs- und Endwert miteinander verglichen, kann die gerichtete Ableitung  $df$  formuliert werden zu Gleichung 2.20.

$$df_{\text{pos}} = f(x_1) \wedge \bar{f}(x_0) = f(x) \wedge \bar{f}(\bar{x}) \quad (2.20)$$

$$df_{\text{neg}} = \bar{f}(x_1) \wedge f(x_0) = \bar{f}(x) \wedge f(\bar{x}) \quad (2.21)$$

Die gerichtete Ableitung trennt also die Boolesche Ableitung in Transitions von 0 nach 1 und 1 nach 0 auf, indem die Anteile des XOR aufgetrennt werden.

### 2.1.8 Darstellung von binären Funktionen

Boolesche Funktionen können zum einen, wie in Kapitel 2.1.1 gezeigt, als Aussage beschrieben werden. Für die Simulation (Validierung) offensichtlicher ist es jedoch, aus der Aussage eine Wahrheitstabelle zu generieren. Die Wahrheitstabelle zeigt die Eingangsbelegungen auf und den, den Belegungen zugeordneten, Ausgangswert. In dieser Dissertation werden totale Wahrheitstabellen immer im gleichen Schema aufgestellt, nämlich so, dass beginnend bei der Null jede Zeile einer aufsteigenden binär kodierten Integer-Zahl zugeordnet werden kann. Eine Beispielwahrheitstabelle ist in Tabelle 2.2 zu sehen.

#	$x_2$	$x_1$	$x_0$	$y$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Tabelle 2.2: Beispielwahrheitstabelle

Die Wahrheitstabelle kann auch als Karnaugh-Veitch-Diagramm (KVD) [15] dargestellt werden, um die Tabelle zu veranschaulichen. Das KVD mit bis zu vier Variablen, hat die schöne Eigenschaft, dass sich jede benachbarte Kachel nur in einer Variablen ändert und so Nachbarschaftsbeziehungen zu sehen sind, welche genutzt werden können, um Resolventen zu bestimmen. Das zu der Tabelle 2.2 gehörige KVD ist in Abbildung 2.6 dargestellt.

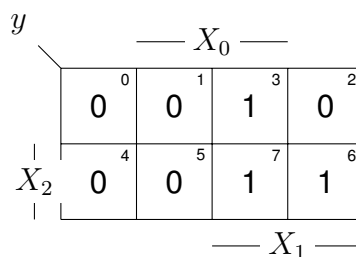


Abbildung 2.6: Beispiel eines KV-Diagramms

Mit der totalen Tabelle kann auch ein Schaltalgebraischer Ausdruck formuliert werden, indem die Einsen der Wahrheitstabelle disjunkt aneinander gereiht und im zweiten

Schritt mit den vorher beschriebenen Rechenregeln vereinfacht werden, um Hardwareaufwand zu minimieren:

$$y = \bar{x}_2 x_1 x_0 \vee x_2 x_1 \bar{x}_0 \vee x_2 x_1 x_0 = x_1 x_0 \vee x_2 x_1 \quad (2.22)$$

Der AA gibt lediglich die Einsen der Wahrheitstabelle wieder und aus diesem kann ein Signalflussplan gezeichnet werden. Der aus dem Schaltalgebraischen Ausdruck entstandene Signalflussplan ist in Abbildung 2.7 dargestellt. Es ist zu erkennen, dass

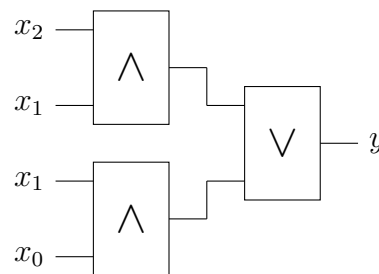


Abbildung 2.7: Signalflussplan des Booleschen Verbandsausdrucks

durch die Konkretisierung der 1, die 0 verloren geht. Entwirft man eine totale Schaltung kann  $\bar{X}^1 = X^0$  vorausgesetzt werden und so die 0 konkretisiert werden. Handelt es sich jedoch um eine reale Schaltung, die i.A. nicht total vorliegen müssen, trifft diese Aussage nicht immer zu. Für partielle Schaltungen gilt also die Beziehung  $\bar{X}^1 = X^0$  nicht.

## 2.2 Ternäre Vektor Listen

Ternärvektorlisten (TVL) werden unter anderem zur rechnergestützten Behandlung Boolescher Funktionen benutzt. Diese Listen bestehen aus Vektoren  $v_i = (v_{n-1}, v_{n-2}, \dots, v_1, v_0)$  der Länge  $n$ , welche aus ternären Variablen bestehen [16]. Einzelne Variablen  $v_i$  haben ihren Wert  $t$  aus dem Wertebereich  $t \in \{0, 1, -\}$ . Der Wert  $-$  beinhaltet beides, also 1 und 0. So kann der Ausdruck  $y = \bar{x}_1 x_0$  des Kodierungsumversums  $x=(x_2, x_1, x_0)$  durch den Ternärvektor (TV)  $v \begin{bmatrix} - & 0 & 1 \end{bmatrix}$  dargestellt werden. Mehrere TV bilden eine TVL. Wenn nicht anders notiert, gilt, dass untereinander geschriebene TV in einer TVL disjunkt vorliegen. So bilden z.B. die 1-er einer Funktion als TVL die boolesche Funktion ab, welcher weitergehend auch mit der TVL resolviert werden kann, um möglichst große Blöcke zu erhalten. So kann Rechenplatz bzw. tatsächliche Fläche von Schaltungen gespart werden. Ziel der Resolvierung ist es, Konsensi zu finden, sodass zwei Vektoren mit einem  $-$  zusammengefasst werden können.

Zwei TV können, wenn sie genau einen Konsensus besitzen, als ein TV zusammengefasst werden, indem der Konsensus durch ein – ersetzt wird. Die Resolvierung soll an einem kurzen Beispiel veranschaulicht werden:

$$\begin{bmatrix} 1 & - & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & - & 1 \\ 0 & - & 1 \end{bmatrix} = \begin{bmatrix} - & - & 1 \end{bmatrix}$$

Es handelt sich um eine TVL mit drei Vektoren und drei Dimensionen in der Form  $TVL = (t_0, t_1, t_2)$ . Zunächst kann ein Consensus zwischen den Vektoren  $t_1$  und  $t_2$  gebildet werden. Der daraus entstandene Vektor  $t_3$  weist mit dem Vektor  $t_0$  einen Consensus auf. So bleibt am Ende der neu gebildete Vektor  $t_4$  übrig. Die Umformung hat ergeben, dass die gesamte TVL nur von der Dimension  $t_0$  abhängt.

### 2.3 Kombinatorische Schaltungen

Binäre Funktionen können als kombinatorische Schaltung realisiert werden [10], [17]. Hierfür kann die allgemeine kombinatorische Schaltung  $S$  definiert werden, als

$$S = (X, Y, f)$$

wobei:

- $X$  ein endliches Eingabealphabet,
- $Y$  ein endliches Ausgabealphabet,
- $f : X \rightarrow Y$  die Ausgabefunktion, die jedem Eingabesymbol einen Ausgang zuordnet, ist.

Eine allgemeine kombinatorische Schaltung ist in Abbildung 2.8 gegeben.

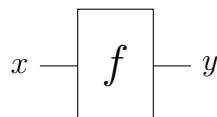


Abbildung 2.8: Allgemeine kombinatorische Schaltung

## 2.4 Darstellung von Automaten

Sequentielle Schaltungen unterscheiden sich zu kombinatorischen Schaltungen dadurch, dass sie vom vorherigen Zustand abhängig sind. Es entsteht also eine zeitliche Abhängigkeit in Form einer Rückkopplung, es entsteht ein Automat. Ein endlicher Automat  $A$  ist ein mathematisches Modell einer sequentiellen Schaltung [18], das zur Beschreibung eines Systems mit endlicher Zustandsmenge verwendet wird. Ein endlicher Automat kann als Quintupel beschrieben werden, siehe Gleichung 2.23.

$$A = (X, Z, Y, \delta, \lambda) \quad (2.23)$$

wobei:

- $X$  ein endliches Eingabealphabet,
- $Z$  eine endliche Menge von Zuständen,
- $Y$  ein endliches Ausgabealphabet,
- $\delta : Z \times X \rightarrow Z$  die Übergangsfunktion, die jedem Zustand und Eingabesymbol einen Folgezustand zuordnet,
- $\lambda : (Z, X) \rightarrow Y$  die Ausgabefunktion ist.

In der Literatur, wie z.B. in [19], wird oft noch ein Startzustand  $Z_i$  angegeben. Dieser wird in dieser Thesis, wenn nicht anders daraufhin gewiesen, der Zustand  $Z_0$  sein. Die Zustandsüberföhrungsfunktion (ZÜF) bildet aus den Eingangsvektoren und den Zustandsvektoren die Zustandsübergänge ab, also zeigt die neuen Zustandsvektoren auf. Die Ausgabefunktion hingegen bildet mit den Zustandsvektoren und, wenn vorhanden, mit den Eingangsvektoren die Ausgabevektoren ab. Hier gibt es auch noch eine Unterscheidung, je nachdem ob der neue Zustand oder der alte Zustand zur Ausgabe führt, in verschiedene Typen, die in Kapitel 2.4.5 näher erläutert wird. Durch die unterschiedliche Ausprägung der Zustandsüberföhrungs- und Ausgabefunktion kommt es zu unterschiedlichen Automatenmodellen. Im Folgenden werden kurz die einzelnen Automatenmodelle beschrieben und am Ende ein kurzer Vergleich der einzelnen Automaten angestrebt.

### 2.4.1 Autonomer Automat

Der autonome Automat  $A = \{Z, \delta\}$  mit der Menge der Zustandsvariablen  $Z$  besitzt eine vom letzten Zustand abhängige ZÜF  $\delta(z)$ . Ein autonomer Automat ist in Abbildung 2.10 dargestellt.

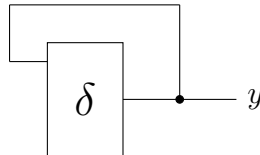


Abbildung 2.9: Autonomer Automat

### 2.4.2 Medwedew-Automat

Der Medwedew-Automat [20]  $A = \{X, Z, \delta\}$  mit der Menge der Eingangsvariablen  $X$ , der Menge der Zustandsvariablen  $Z$  besitzt eine vom Eingang und letztem Zustand abhängige ZÜF  $\delta(z, x)$  und kann als Spezialform eines Moores betrachtet werden, bei dem die Ausgabefunkt  $\mu$  gleich 1 ist. Ein Medwedjew-Automat ist in Abbildung 2.10 dargestellt.

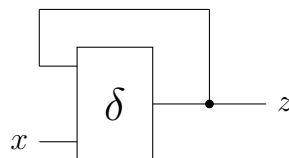


Abbildung 2.10: Medwedjew-Automat

### 2.4.3 Moore-Automat

Der Moore-Automat [21]  $A = \{X, Y, Z, \delta, \mu\}$  mit der Menge der Eingangsvariablen  $X$ , der Menge der Ausgangsvariablen  $Y$ , der Menge der Zustandsvariablen  $Z$  besitzt eine vom Eingang und letztem Zustand abhängige ZÜF  $\delta(z, x)$  und eine Ausgabefunktion  $\mu(z)$ , die lediglich vom Zustand abhängig ist. D.h. jeder Zustand führt zu einem anderen Ausgang. Ein Moore-Automat ist in Abbildung 2.11 dargestellt.

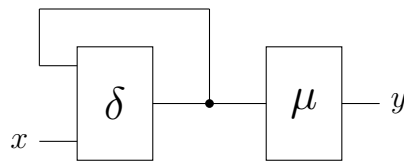


Abbildung 2.11: Moore-Automat

#### 2.4.4 Mealy-Automat

Der Mealy-Automat [22]  $A = \{X, Y, Z, \delta, \lambda\}$  mit der Menge der Eingangsvariablen  $X$ , der Menge der Ausgangsvariablen  $Y$ , der Menge der Zustandsvariablen  $Z$  besitzt eine vom Eingang und letztem Zustand abhängige ZÜF  $\delta(z, x)$  und eine Ausgabefunktion  $\lambda(z, x)$ , die vom Zustand und vom Eingang abhängig ist. Einzelne Zustände können also zu verschiedenen Ausgaben führen. Ein Mealy-Automat ist in Abbildung 2.12 dargestellt.

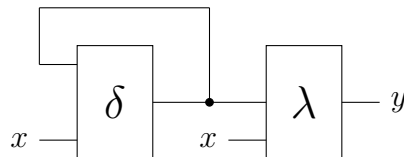


Abbildung 2.12: Mealy-Automat

#### 2.4.5 Typ alt und Typ neu

Wie Anfang des Kapitels beschrieben, können Automaten auch vom zeitlichen Abgreifen des Zustands unterschieden werden. Die Kategorisierung des Automaten als Typ alt oder neu erfolgt, indem bestimmt wird, ob die Ausgabefunktion vor der nach der Synchronisation abzapft. In Abbildung 2.13 ist der Moore-Automat als Typ neu und als Typ alt dargestellt. Es kann ohne Beschränkung der Allgemeinheit gesagt werden,

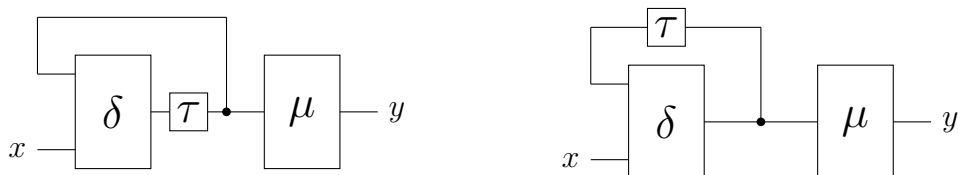


Abbildung 2.13: Moore-Automat Typ alt und Typ neu

dass taktlose Schaltungen vom Typ neu sind, da diese keinen Synchronisationspunkt haben.

### 2.4.6 Vergleich der Automaten

Im Folgenden wird der Mealy-Automat mit dem Moore-Automaten verglichen. Der Medwedew-Automat wird nicht gesondert betrachtet, da er als Spezialfall des Moore-Automaten angesehen werden kann. Ebenso bleibt der autonome Automat unberücksichtigt, da im weiteren Verlauf der Arbeit Ereignisse – also Eingangsbelegungen des Automaten – von Bedeutung sind und ein autonomer Automat daher nicht mehr relevant ist. Der Vergleich ist in Tabelle 2.3 zusammenfassend dargestellt.

Eigenschaft	Mealy	Moore
Anzahl Zustände	↓	↑
Abhängigkeit des Ausgangs vom Eingang	↑	↓
Komplexität der Ausgabefunktion	↑	↓
Sicherheit	↓	↑

Tabelle 2.3: Vergleich von Mealy- und Moore-Automaten

Es werden im Verlauf dieser Arbeit verschiedene Automatentypen entworfen und die Sicherheit des Mealy-Automaten gesondert betrachtet. Ziel ist es, sichere Mealy-Automaten entwerfen zu können, um Fläche zu sparen, indem weniger Zustände verwendet werden müssen.

### 2.4.7 Z-Gleichungen und Automatengraphen

Um die Zustandsüberführungs-Gleichungen aufzustellen, müssen zunächst alle Elementarkonjunktionen  $k_i(x)$  mit den Kanten, bzw. dem Übergangsausdruck,  $h_{ij}$  konjunktiv verknüpft werden, welche zum Zustand  $Z_j$  führen. Die disjunktive Verknüpfung dieser Konjunktionen ergibt die Zustandsüberführungsgleichungen. Als Formel aus [11]:

$$k_j(z) := \bigvee_{i=0}^{2^p-1} k_i(z) \wedge h_{ij}(x) \quad (2.24)$$

wobei gilt:  $(\mathcal{W}(h_{ij}, X_k) = 1) \Leftrightarrow (\delta(Z_i, X_k) = Z_j)$  mit  $X_k \in X$ .

Da für die technische Realisierung die einzelnen Zustandsvariablen, also beispielsweise im Kapitel Durchführung  $[z_2, z_1, z_0]$ , bestimmt werden müssen, werden für die Zustandsvariablen die sogenannten z-Gleichungen [11] aufgestellt durch:

$$z_s := \bigvee_{j \in M_s} \left( \bigvee_{i=0}^{2^p-1} k_i(z) \wedge h_{ij}(x) \right) \quad (2.25)$$

mit  $M_s = \{j \mid Z_j(z_s) = 1\}$ .

Der Automatengraph eines Moore-Automaten hat die Form  $G_\mu = [Z, K, \omega_\delta, \omega_\mu]$  mit den Zuständen als Knotenmenge  $Z$  des Graphen, mit den Transitionen als Kantenmenge  $K$ , der Kantengewichtsfunktion  $\omega_\delta$ , den Transitionsbedingungen bzw. dem Übergangsausdruck, und der zustandsbezogenen Ausgabefunktion  $\omega_\mu$ . Die Knoten werden als Kreise und die Kanten als Pfeile dargestellt. Um einen AG auf seine Vollständigkeit zu überprüfen, d.h. dass alle Belegungen in den wegführenden Kanten und den Eigenschleifen vorhanden sind, werden die folgenden Bedingungen in [11] formuliert:

$$\forall i \left( \bigvee_{j=0}^{2^p-1} h_{ij}(x) = 1 \right) \quad (2.26)$$

Auch für die Widerspruchsfreiheit, also dass es z.B. nicht zwei Kanten gibt, die bei gleicher Eingangsbelegung vom gleichen Zustand in verschiedene Richtungen führen, gibt es die Formel aus [11] zur Überprüfung, indem die wegführenden Kanten konjunkt werden und gleich null ergeben müssen, was bedeutet, dass die Kanten disjunkt zueinander sind:

$$\forall i \left( \bigvee_{j,k=0; j \neq k}^{2^p-1} h_{ij}(x) \wedge h_{ik}(x) = 0 \right) \quad (2.27)$$

## 2.4.8 Stabilisierungsarten eines Automaten

Im Folgenden werden verschiedene Stabilisierungsarten eines rückgekoppelten Systems erläutert.

### 2.4.8.1 Zustandsstabilisierung

Die Zustandsstabilisierung erfolgt mit dem RS-Buffer (RSB) [23]. Das Schaltbild auf TL ist in Abbildung 2.14 dargestellt. Zunächst werden die Eingangssignale  $s$  und  $r$  betrachtet. Sind die Signale zueinander invertiert, wird der Tristate geschaltet, bei  $(s, r) = [10]$  ist der Ausgang  $b = 1$  sowie für  $(s, r) = [01]$  der Ausgang  $b = 0$ . Liegt ein Eingangssignal  $s \sim r$  vor, d.h.  $s$  und  $r$  sind äquivalent zueinander, sieht man am Ausgang des Tristate  $\bar{X}$  high-Z und der Babysitter behält den alten Zustand bei. Somit ist der Ausgangspin B von den Eingangssignalen isoliert. Gleichung. 2.28 stellt einen Teil des

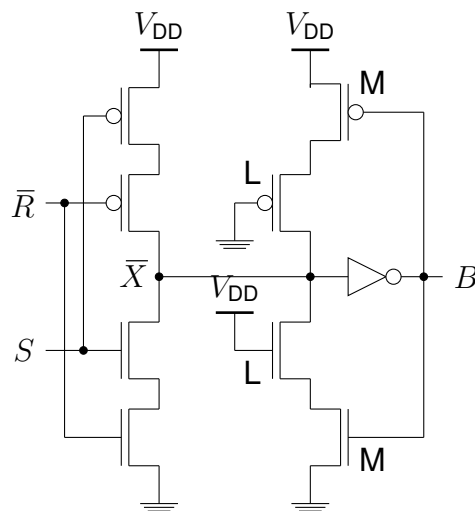


Abbildung 2.14: RS Buffer in TL

Babysitters dar.

$${}^a z = \delta({}^a z, *) \quad (2.28)$$

Die Zustandsstabilisierung wirkt so, dass erstens bei zu geringer Energie des Eingangssignals (d.h. Setup- und Hold-Zeiten wurden nicht eingehalten) der Tristate nicht schaltet. Zweitens, wenn Pfadverzögerungen auftreten, d.h. wenn  $s \sim r$  für eine kurze Zeit gültig ist, schaltet der Tristate auf High-Z und der alte Zustand wird gehalten.

### 2.4.8.2 Funktionsstabilisierung

Funktionsstabilisierung kann unter zwei Gesichtspunkten vorgesehen werden. Zum einen gibt es diese Stabilisierung in der Funktion (also Eigenschleife im SFG z.B.), zum anderen muss aber auch sicher gestellt werden, dass die Schaltung auch in diesem Zustand verbleiben kann, also die Eigenschleife sich maximal festfrisst, damit keine Transienten entstehen, dies wurde als Funktionsstabilität in der Struktur betitelt.

#### Funktionsstabil in der Funktion

Funktionsstabilisierung bedeutet, dass jede eingehende Kante (Kantenereignis) durch sich selbst (gleiches Kantenereignis) stabilisiert wird (grafisch kann eine Selbstschleife gezeichnet werden) [10], [11]. Dieser Knoten wird dann als funktionsstabilisierter Zustand bezeichnet. Diese Selbstschleife muss tatsächlich implementiert werden. Die Zu-

standsstabilisierung ist axiomatisch, daher muss sie in der SFG gar nicht eingezeichnet werden. Die Selbstschleife der Funktionsstabilisierung muss durch eine Selbstschleife angegeben werden. Die Kante  $\bar{A}$  führt vom Knoten  $Z_0$  in den Zustand  $\bar{Z}_0$ , da dieser

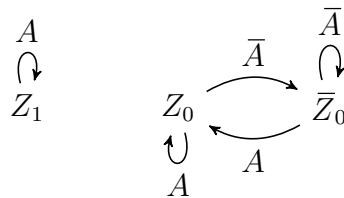


Abbildung 2.15: SFG eines Funktionsstabilisierten Automaten (zusätzlich ein funktionsstabilisierter isolierter Knoten)

die gleiche Kante wie eine Selbstschleife besitzt. Das gleiche gilt für die Kante  $A$  vom Zustand  $\bar{Z}_0$  in den Knoten  $Z_0$ . In Formeln ausgedrückt, ist die Bedingung für die Funktionsstabilisierung (in 2nd Order Logic) in Gleichung 2.29 gegeben.

$${}^a z = \delta({}^a z, x) \quad (2.29)$$

wobei  $z$  der Zustand und  $x = (a)$  ist.

In [11] wird die Überprüfung der Stabilitätsbedingung an den Automatengraphen angepasst, mit der vom Zustand  $Z_i$  zum Zustand  $Z_j$  führenden Kante  $h_{ij}$  und der Eigenschleifenkante  $h_{jj}$  des Zustandes  $Z_j$ :

$$\bigvee_{i \neq j} h_{ij} \wedge \bar{h}_{jj} = 0 \quad (2.30)$$

Der Zustand  $Z_{jj}$  ist stabil, wenn die obere Bedingung erfüllt ist.

### Funktionsstabil in der Struktur

Damit eine Schaltung in der Struktur funktionsstabil ist, muss der Rückkoppelpfad eine geringere Verzögerung als der Vorwärtspfad aufweisen, die Bedingung für die Schaltung in Abb. 2.16 muss also  $\tau_{\delta} > \tau_{\Delta}$  [24] sein.

#### 2.4.8.3 Strukturstabilisierung

Bei der Strukturstabilisierung wird die Schaltung durch eine lokale Stabilisierung, z. B. durch Speicherelemente, ergänzt und stabilisiert, indem kritische Pfade der Schal-

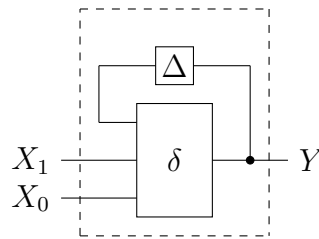


Abbildung 2.16: Funktionsstabile Schaltung

tung gepuffert und durch ein Taktsignal ausgelöst werden [25]. Eine strukturstabilisierte Schaltung ist also eine Schaltung, die durch ein angelegtes Taktsignal synchronisiert wird, und die Strukturstabilisierung wird durch einen Kreis im SFG angezeigt. Die Formel für die Strukturstabilisierung ist in Gl. 2.31 angegeben.

$${}^a z := \delta({}^a z, x) \quad (2.31)$$

## 2.5 Dynamische Effekte digitaler Schaltungen

Werden Schaltungen synchron betrieben, wird der Takt an die dynamische Veränderung angepasst, d.h. der Schaltung wird von außen genug Zeit gegeben, bis diese sich einschwingt bzw. festfrisst. Werden asynchrone Schaltungen realisiert müssen diese sich entweder selbst sperren, bis sie eingeschwungen sind oder solche Effekte dürfen gar nicht erst auftreten. Hierfür werden in diesem Abschnitt die Modelle der dynamischen Effekte etwas genauer erläutert und deren mögliche Behebungen dargestellt. Digitale Schaltungen funktionieren in der Theorie anhand ihrer Wahrheitstabelle und der synthetisierten Struktur wie geplant. Oft zeigen auch Simulationen das logische Verhalten an. In der Realität sieht das jedoch u.U. anders aus. Wenn also von der logischen Ebene in die Strukturebene eingetaucht wird, können Verhaltensweisen der Schaltung auftreten, die so nicht gewünscht und nicht vorhergesehen waren. Bei asynchronen Schaltungen sind dies vor allem die zwei Kategorien der Hazards und Races. Die Fehlermodelle können wie in Abbildung 2.17 eingebettet werden. Das Diagramm veranschaulicht, wie die angegebenen Auswirkungen zu differenzieren sind. Zunächst werden die Races von den Strukturhazards exkludiert und dann die verbleibenden Strukturhazards von den Funktionshazards herausgenommen, um die einzelnen Auswirkungen separat zu untersuchen. Races werden durch ihre Wettläufe auf den Rückkoppelleitungen differenziert, Strukturhazards, indem Funktionshazards ausgeschlossen werden (Schalten von nur einer Eingangsvariablen). In diesem Abschnitt

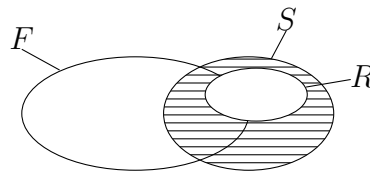


Abbildung 2.17: Venn-Diagramm zur Einordnung von Funktionshazards ( $F$ ), Strukturhazards ( $S$ ) und Races ( $R$ )

werden kurz die theoretischen Grundlagen für Hazards, Races und Glitches erläutert, um zu verstehen, wie sie logisch betrachtet werden können. Die in dieser Arbeit verwendeten Definitionen und die Nomenklatur basieren weitestgehend auf [10] und [26], es wird aber auch zusätzlich auf [27]–[32] verwiesen.

### 2.5.1 Hazards

Hazards beschreiben die Möglichkeit, dass beim Wechsel eines Eingangssignals kurzzeitige unerwünschte Änderungen am Ausgang einer elektronischen Schaltung auftreten können. Wenn ein Hazard tatsächlich am Ausgang auftritt, wird er als Hazardfehler bezeichnet, da er die korrekte Signalverarbeitung kurzzeitig verfälschen kann [10]. Hazards können basierend auf ihrer Herkunft in Funktions- oder Strukturhazards unterteilt werden sowie nach ihrer Wirkung in statische oder dynamische Hazards. In Abbildung 2.18 ist exemplarisch ein statischer 1-Hazard eines Ausgangs  $y$  über der Zeit  $t$  dargestellt. Statische Hazards bewirken eine kurzzeitige Änderung des Aus-

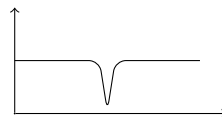


Abbildung 2.18: Statischer Funktionshazard

gangssignals, bevor es zu seinem ursprünglichen Signalpegel zurückkehrt. Dynamische Hazards hingegen führen während des Übergangs auf einen neuen Ausgangspegel vorübergehend zu einer Rückkehr auf den vorherigen Wert. Die mathematische Beschreibung und Herleitung von Hazards erfolgt hier gemäß der Darstellungen in [10] und [26].

#### 2.5.1.1 Funktionshazards

Ein Funktionshazard beschreibt die Möglichkeit eines kurzzeitigen Fehlers am Ausgang einer Schaltung, der durch das ungleichzeitige Umschalten von mindestens zwei

Eingangsvariablen verursacht wird. Wenn beim Übergang von einer Eingangskombination zur nächsten eine Zwischeneingangskombination auftritt, kann dies dazu führen, dass der Ausgang vorübergehend einen falschen Wert annimmt. Funktionshazards entstehen aus der Funktion selbst und sind unabhängig von der physikalischen Struktur der Schaltung. Dabei wird die Schaltung als idealisierte, verzögerungsfreie Komponente betrachtet.

Ein Beispiel für einen dynamische Funktionshazard ist in Abbildung 2.19 zu sehen. Auch ein möglicher Weg im KV-Diagramm einer kombinatorischen Schaltung  $Y$  mit den Eingängen ( $X_2, X_1, X_0$ ) zum dynamischen Hazard ist eingezeichnet.

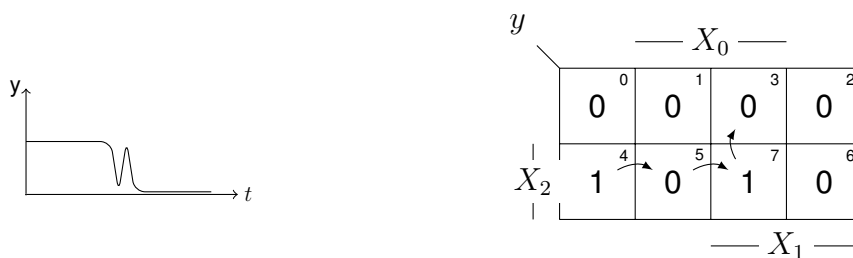


Abbildung 2.19: Beispiel eines dynamischen Hazards

Im Beispiel wird aus Belegung 4 ([100]) in Belegung 3 ([011]) geschaltet. Dabei ändert sich zuerst der Wert von  $x_0$ , sodass der Ausgang kurzzeitig auf 0 geht, dann ändert sich der Wert von  $x_1$ , der den Ausgang auf 1 setzt und dann wiederum schaltet  $x_2$  und setzt den Ausgang auf den gewünschten Wert 0.

### Formelle Beschreibung von Funktionshazards

Die notwendige Bedingung für das Vorliegen eines statischen Hazards besteht darin, dass der gleiche Ausgangswert vorliegt, wenn alle Eingänge gleichzeitig schalten, also dass die vektorielle Ableitung gemäß Gleichung 2.32 null ist.

$$\left. \frac{df(x)}{dx_p} \right|_{x_p} = f(x_p, x_q) \approx f(x_p \approx 1, x_q) \text{ mit } x = (x_p, x_q) \quad (2.32)$$

Die hinreichende Bedingung für die Entstehung eines statischen Hazards ist, dass ein nicht-simultanes Umschalten der Eingangssignale, zu unterschiedlichen Ausgangswerten führt. Dies entspricht der Bedingung, dass die Variationsableitung einen Wert 1

annimmt, wie in Gleichung 2.33 ausgedrückt.

$$\left. \frac{\Delta f(x)}{\Delta x_p} \right|_{x_p} = \bigvee_c f(x_p, x_q) \approx f(x_p \approx c, x_q) \quad (2.33)$$

mit  $x_p = (x_1, x_0)$  und  $c \in \{01, 10, 11\}$

Die allgemeine Formel zur Erkennung eines jeden statischen Funktionshazard bedient sich dem Differential, siehe Gleichung 2.18 und der Variation, siehe Gleichung 2.19, und lautet:

$$\approx_{sF} = \overline{d\phi} \delta \phi \quad (2.34)$$

Dynamische Funktionshazards können während eines Übergangs eines Ausgangssignals auftreten, wenn mindestens drei Eingangssignale gleichzeitig ihren Wert ändern.

Die Existenzbedingung für einen dynamischen Funktionshazard liegt vor, wenn zwei unterschiedliche Eingangskombinationen zu unterschiedlichen Ausgangswerten führen. Diese Bedingung findet ihre Entsprechung in der mathematischen Beschreibung durch die vektorielle Ableitung der das System beschreibenden Gleichung, die gemäß Gleichung 2.35 den Wert 1 ergibt.

$$\left. \frac{df(x)}{dx_p} \right|_{x_p} = 1 \quad \text{mit } x = (x_p, x_q) \quad (2.35)$$

Zur Veranschaulichung dieser Thematik betrachten wir  $\underline{x}$  als einen Vektor, der aus vier Elementen besteht. Damit ein dynamischer Hazard auftritt, muss eine Eingangssignalvariation durch nicht-simultanes Umschalten von  $X_0$  zu  $X_3$  auftreten, die unterschiedliche Ausgangssignale erzeugt. Dies lässt sich mathematisch wie folgt ausdrücken:  $f(X_0) = \bar{f}(X_1) = f(X_2) = \bar{f}(X_3)$ .

Ein dynamischer Funktionshazard setzt sich aus zwei statischen Funktionshazards zusammen und kann somit formell durch die obigen Gleichungen hergeleitet werden.

### Vermeidung von Funktionshazards

Zur Vermeidung von Funktionshazards lassen sich nach [11] und [10] folgende Optionen definieren:

**Definition 8** *Vermeidung von Funktions hazards*

- Es dürfen nur Belegungswechsel von genau einer Eingangsvariablen zugelassen werden.
- Das Ausgangssignal muss so verzögert werden, dass eine Zwischenbelegung keine Änderung verursacht.
- Die Belegungswechsel müssen mithilfe eines Taktes synchronisiert werden.
- Änderung der Funktion, sodass keine Wertkombination bei einer Eingangsänderung zu einem Hazard führt.
- Umkodierung der Eingänge (mit dem gleichen Effekt wie bei der Änderung der Funktion).

Der erste Punkt bezieht sich auf die Einschrittigkeit bei der Änderung der  $x$ -Variablen und wird im weiteren Verlauf der Arbeit zur Realisierung von delay-insensitive Schaltungen angewandt. Der zweite Punkt beschreibt eine nachträgliche Optimierung von Schaltungen, stellt jedoch keine allgemeingültige Lösung dar und wird daher nicht weiterverfolgt. Der dritte Punkt, die Synchronisation der Belegungswechsel mithilfe eines Taktes, findet sich bei asynchronen Schaltungen in Bundled-Data-Protokollen wieder und wird im weiteren Verlauf bei den selbstsperrenden Dominologik-Schaltungen betrachtet. Die letzten beiden Punkte werden beispielsweise in [31] im Sinne des Burst-Modus verwendet, bleiben jedoch außerhalb des Fokus dieser Arbeit.

**2.5.1.2 Struktur hazards**

Struktur hazards beschreiben die Möglichkeit des Auftretens eines fehlerhaften Signals am Ausgang einer Schaltung, das durch unterschiedliche Verzögerungen entlang verschiedener Signalpfade verursacht wird. Im Gegensatz zu Funktions hazards wird bei Struktur hazards die Latenz in der Struktur berücksichtigt. Um Struktur hazards eindeutig von Funktions hazards unterscheiden zu können, werden sie so betrachtet, dass die Eingänge gleichzeitig schalten, um Funktions hazards auszuschließen. In den folgenden Betrachtungen werden Funktions hazards ausgeschlossen, indem nur einzelne Eingangsvariablen gewichtet werden, wodurch keine Funktions hazard auftreten können, da diese das Schalten von mindestens zwei Eingangsvariablen voraussetzen.

## Strukturhazards zweistufiger Schaltungen

Bei zweistufigen Schaltungen können Strukturhazards auftreten, selbst wenn nur ein Eingangssignal schaltet. Ein typisches Beispiel ist eine Weggabelung, gefolgt von einem Zusammentreffen zweier Signalpfade, von denen einer eine Negation enthält.

In diesem Fall wird das Eingangssignal  $x$  durch ein Negationsglied in einem anderen Block verzögert, sodass für kurze Zeit sowohl  $X$  als auch  $\bar{X}$  gleichzeitig anliegen. Abbildung 2.20 zeigt eine Struktur, die einen solchen Strukturhazard aufweist, sowie das zugehörige KV-Diagramm.

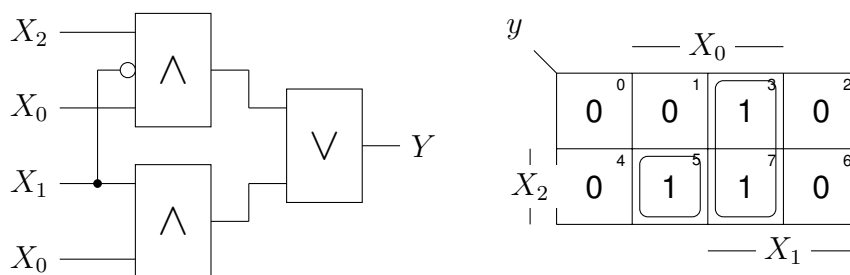


Abbildung 2.20: Struktur mit potentiellm Strukturhazard

Zu sehen ist, dass beide AND-Gatter disjunkt zueinander sind, da einmal  $X_1$  und einmal  $\bar{X}_1$  vorkommt. Wenn nun von der Belegung 7 zur Belegung 5 geschaltet wird - also  $x_1$  von 1 nach 0 schaltet, kann durch Verzögerungen des Inverters von  $\bar{X}_1$ , kurzzeitig  $(X_1, \bar{X}_1) = [00]$  anliegen, wodurch beide And-Gatter logisch 0 sind und somit der Ausgang kurzzeitig auf die logische 0 gezogen wird. Die Schaltung kann strukturhazardfrei realisiert werden, indem beide Blöcke überlappen und so die Funktion vereinfacht wird zu  $y = x_1x_0 \vee x_2x_0$ . Dies vermindert in diesem Fall auch den Schaltungsaufwand.

## Formelle Beschreibung von Strukturhazards

Ein Strukturhazard liegt vor, wenn Funktionshazard ausgeschlossen werden können, also die Variationsbleitung null ist, wie in Gleichung 2.36 dargestellt.

$$\frac{\Delta f(x)}{\Delta x_p} = 0 \quad (2.36)$$

Um Strukturhazards mathematisch zu betrachten, wird das Totzeitmodell [10] der Schaltung aufgestellt. Das Totzeitmodell der Schaltung schiebt alle Leitungsverzögerungen nach vorne und jeder Pfad wird einzeln betrachtet, bekommt also seine eigene Kodierung. Die logische Funktion der Schaltung im Kodierungsuniversum  $x = (x_{i-1}, x_{i-2}, \dots, x_1, x_0)$  wird nun in das Kodierungsuniversum  $a = (a_{j-1}, a_{j-2}, \dots, a_1, a_0)$  mit  $i \leq j$  eingebettet. Das allgemeine Totzeitmodell ist in Abbildung 2.21 gegeben. Das Vorliegen von Strukturhazards impliziert, dass beim Umschalten des in  $a$  kodier-

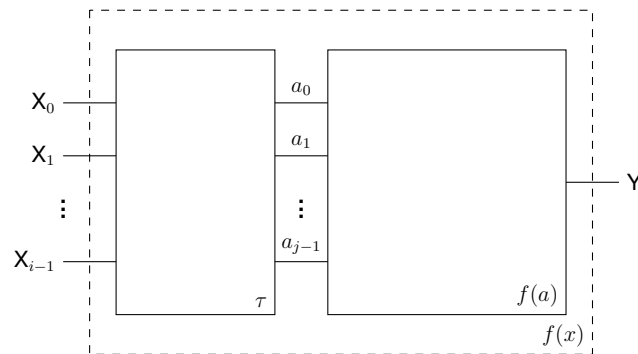


Abbildung 2.21: Allgemeines Totzeitmodell

ten Vektors Variationen auftreten. Dies äußert sich in einer Variationsableitung, die den Wert 1 annimmt, wie in Gleichung 2.37 dargestellt.

$$\frac{\Delta f(a)}{\Delta a_p} = 1 \text{ mit } a = (a_p, a_q) \quad (2.37)$$

Dies bedeutet, dass sich der Funktionswert durch Laufzeitunterschiede kurzzeitig ändert und somit ein statischer Strukturhazard vorliegt. Auch dynamische Strukturhazards können entstehen. Da in zweistufigen Strukturen aber nur statische Strukturhazards entstehen, siehe [11], speziell für DNF- und NAND-Strukturen statische 1-Hazards, wird auf die dynamischen Strukturhazards nicht weiter eingegangen. Um Strukturhazards zu vermeiden, können zudem redundante Blöcke zur Schaltung hinzugefügt werden, die sogenannten Consensi, die nicht überlappende Blöcke verbinden, indem sie bei beiden überlappen. Hieraus lassen sich Optionen zur Vermeidung von Strukturhazards definieren:

**Definition 9** *Vermeidung von Strukturhazards*

- Änderung der Struktur: Um Strukturhazards zu vermeiden, sollten die einzelnen Schaltungsblöcke überlappend sein (auch unter dem Nachteil einer Redundanz).
- Taktbetrieb der Eingänge (wie bei den Funktionshazards).
- Einfügen von Verzögerungen (wie bei den Funktionshazards).
- Vermeidung von Strukturhazards durch kaskadierte Funktionen

**2.5.2 Races**

Feedback wird in der Schaltungsanalyse im Rückkopplpfad aufgeschnitten und dann kombinatorisch betrachtet. Das Verhalten ist damit dann deterministisch. Allerdings berücksichtigt die Logik keine physikalischen Effekte, und Variationen in der Stromtreiberfähigkeit sowie unterschiedliche Verzögerungen können zu Wettläufen, in dieser Thesis als Races bezeichnet, auf der Rückkopplungsleitung führen. Wird die Schaltung in einen Zustand versetzt, der ein Race auslöst, so bestimmt die Physik das endgültige Ergebnis.

Race-Phänomene lassen sich in zwei Kategorien einteilen [10], [29]. Bei einem nicht-kritischen Race ist der endgültige Ausgangszustand des Systems zwar eindeutig, jedoch existieren mehrere vorübergehende Zwischenzustände, die zum finalen Ausgang führen.

Bei einem sogenannten kritischen oder fatalen Race kann der Zustand auf unvorhersehbare Weise in mehreren unterschiedlichen Kombinationen enden.

In Abbildung 2.22 ist das low-active RS-Latch gezeigt, das als einfachster Vertreter für Race-Situationen betrachtet wird und genutzt wird, um eine allgemeine Race-Formel herzuleiten. Das RS-Latch hat zwei Rückkoppelleitungen, die in Racesituationen miteinander konkurrieren.

Ein fatales Race entsteht beim Umschalten der beiden Eingänge  $\bar{R}$  und  $\bar{S}$  von [00] auf [11]. Bei der Eingangsbelegung [00] befindet sich das RS-Latch mit seinen Zustands-Pins  $Q_1$  und  $Q_0$  im stabilen Zustand [11]. Beim Schalten der Eingänge auf [11] versuchen beide NAND-Gatter auf die logische 0 zu ziehen. Da beide Rückkopplungen aber nie genau gleichzeitig ankommen werden bzw. die Nand-Gatter nicht gleich stark

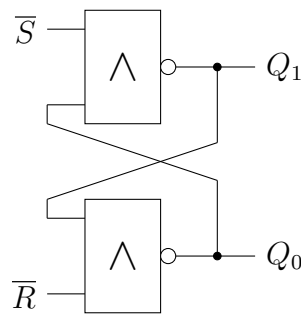


Abbildung 2.22: Das Low-Active RS-Latch

treiben, zieht ein NAND-Gatter zuerst zur logischen 0 und setzt damit den anderen Zustand auf die logische 1. Die logische sowie die reale Wahrheitstabelle sind in Tabelle 2.4 aufgeführt [23].

Ausgangsbelegung				logisch		real	
$Q_1$	$Q_0$	$S$	$R$	$Q_1$	$Q_0$	$Q_1$	$Q_0$
0	0	0	0	1	1	*	*
0	0	0	1	1	1	*	1
0	0	1	0	1	1	1	*
0	0	1	1	1	1	1	1
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	1
0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	1	0
1	0	0	1	1	1	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	1	1	1
1	1	0	0	0	0	*	*
1	1	0	1	0	1	0	1
1	1	1	0	1	0	1	0
1	1	1	1	1	1	1	1

Tabelle 2.4: Wertetabelle mit logischem und realem Zustand

Diese Races sind in der logischen Wertetabelle nicht ersichtlich, in der Realität treten sie aber auf. Da die Werte [00] der Wahrheitstabelle nicht erreichbar sind, müssen diese durch \* ersetzt werden. Das Zeichen \* bedeutet, dass die Belegung aus dem Kodierungsuniversum genommen wird. Ein Race entsteht also, wenn die Reihenfolge des Umschaltens von zwei Zustandsvariablen zu verschiedenen Ausgaben führen kann. Der Startpunkt des Races des RS-Latch zeigt dabei metastabiles Verhalten. Eine Illustration für Metastabilität ist in Abbildung 2.23 dargestellt.

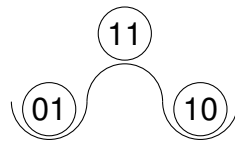


Abbildung 2.23: Veranschaulichung der Metastabilität

Die Kugel wird je nach Umwelteinfluss einmal zur linken und einmal zur rechten Seite rollen. Analog ist auch der Ausgang der Zustände im Beispiel nicht eindeutig vorherzusagen. Um Races zu bestimmen, werden zunächst die KV-Diagramme, siehe Abbildung 2.24, des Low-Active RS-Latches bei der Race-Bedingung betrachtet.

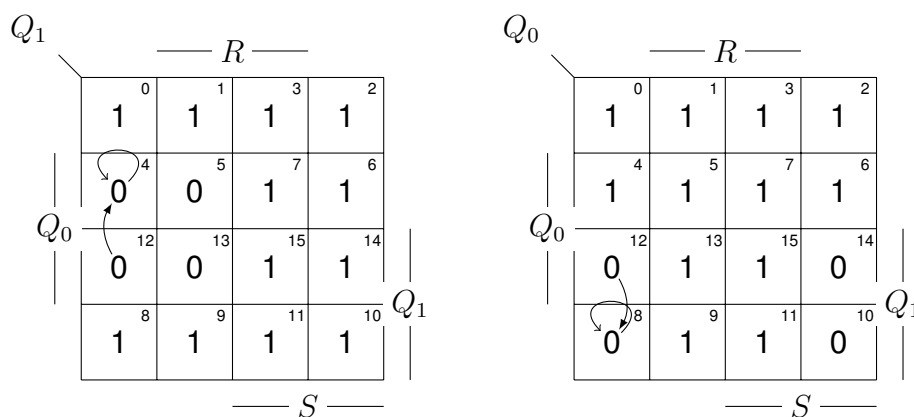


Abbildung 2.24: KV-Diagramme für  $Q_1$  und  $Q_0$

Im KV-Diagramm ist zu sehen, dass in der Belegung 12, also dem Startpunkt des fatalen Races die Zustände jeweils in verschiedene Richtungen laufen und dann in einem Zustand enden der sich selbst stabilisiert. Um eine Gesetzmäßigkeit für das Race zu finden, wird zunächst die Situation des Races genauer betrachtet. Das RS-Latch besteht aus zwei Teilen: einem kombinatorischen und einem sequentiellen Teil. Das wird deutlicher wenn die NAND-Struktur durch DeMorgan umgewandelt wird (beispielhaft für  $Q_1$ ):

$$Q_1 := \overline{Q_0 \wedge \bar{S}} = \bar{Q}_0 \vee S \quad (2.38)$$

$Q_1$  ist also abhängig von einem sequentiellen Anteil  $\bar{Q}_0$  und einem kombinatorischen Anteil, der Identität von  $S$ . Wird nun der Automat durch die Überlagerung des kombinatorischen Anteils in einen Zustand gebracht, der nach Neutralisierung des kombinatorischen Anteils, d.h. beim NAND dem Schalten zur logischen 1, untragbar wird, siehe Abbildung 2.25, kommt es in dieser Schaltungskonfiguration zu einem fatalen Race,

wenn  $Q_1 = Q_0$  gilt. Es ist also deutlich zu erkennen, dass das Race zwei Bedingungen

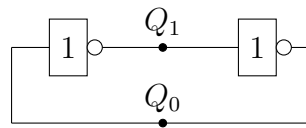


Abbildung 2.25: Untragbare Situation in der Race-Konstellation ( $Q_1 \sim Q_0$ )

in diesem Fall benötigt: Es muss eine Abhängigkeit in Form einer Rückkopplung der beiden Variablen bestehen und die Zustände müssen beide schalten.

Das Race kann auch genauer angeschaut werden. Es wird ein Zustand eingestellt, der durch den kombinatorischen Anteil stabil zu [11] führt, dann werden  $\bar{S}$  und  $\bar{R}$  so umgeschaltet, dass das RS-Latch nur noch abhängig von den Rückkopplungsleitungen ist und diese führen beide eine Transition aus.

### Weiteres Beispiel für ein Race

Gegeben sei ein Medwedjew-Automat  $A=(X, Y, Z, \delta)$  mit  $x = (x_1, x_0)$  und  $z = (z_1, z_0)$  und den ZÜF

$$\delta_{z_0}(z, x) = \bar{x}_0 \vee z_1 x_1 \quad \delta_{z_1}(z, x) = z_0 x_1 x_0 \vee z_1 x_1$$

Die Wahrheitstabelle des Automaten ist in Tabelle 2.5 aufgeführt. Das Schaltbild des Automaten ist in Abbildung 2.26 dargestellt.

Betrachtung der NAND-Struktur von  $\delta$ :

$$\delta_{z_0}(z, x) = \overline{x_0 \wedge \overline{z_1 x_1}} \quad \delta_{z_1}(z, x) = \overline{z_0 x_1 x_0 \wedge \overline{z_1 x_1}}$$

Es gibt auch hier zwei konkurrierende asynchrone Rückkopplungen, welche zu einem Race führen können. Die Race-Ausgangsbedingung und die NAND-Struktur sind in Abbildung 2.27 zu sehen.

Das Race führt nun dazu, dass der Ausgang entweder 00 oder 11 beträgt, siehe Abbildung 2.28.

Aus der Tabelle können direkt die KV-Diagramme gezeichnet, siehe Abbildung 2.29, und die Bedingung des Races näher betrachtet werden. Auch hier ist zu erkennen, dass die zwei Bedingungen vom vorherigen Beispiel erfüllt werden. Zunächst ist im

$z_1$	$z_0$	$x_1$	$x_0$	$\delta_{z_1}$	$\delta_{z_0}$
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	1	1

Tabelle 2.5: Wahrheitstabelle des Automaten

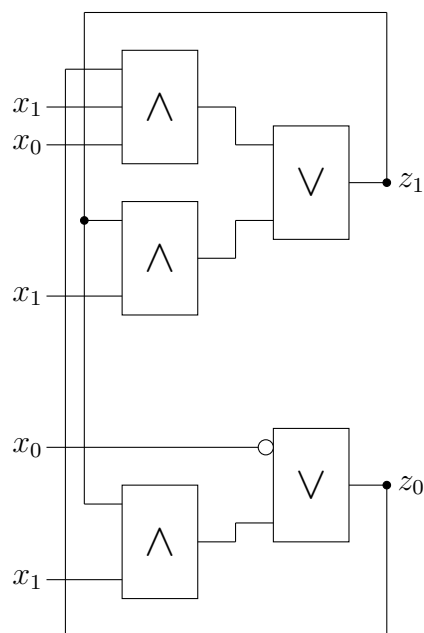


Abbildung 2.26: Schaltplan des Automaten

Fall des Races eine Abhängigkeit der Zustandsvariablen von der jeweiligen anderen gegeben, außerdem liegt eine Doppeltransition nach Umschalten der Eingänge vor.

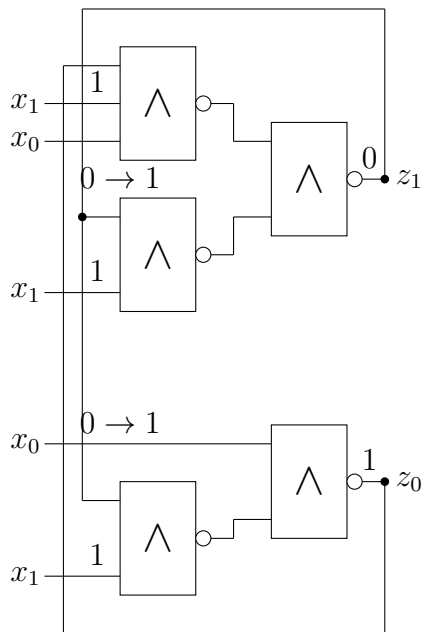


Abbildung 2.27: Race-Bedingung

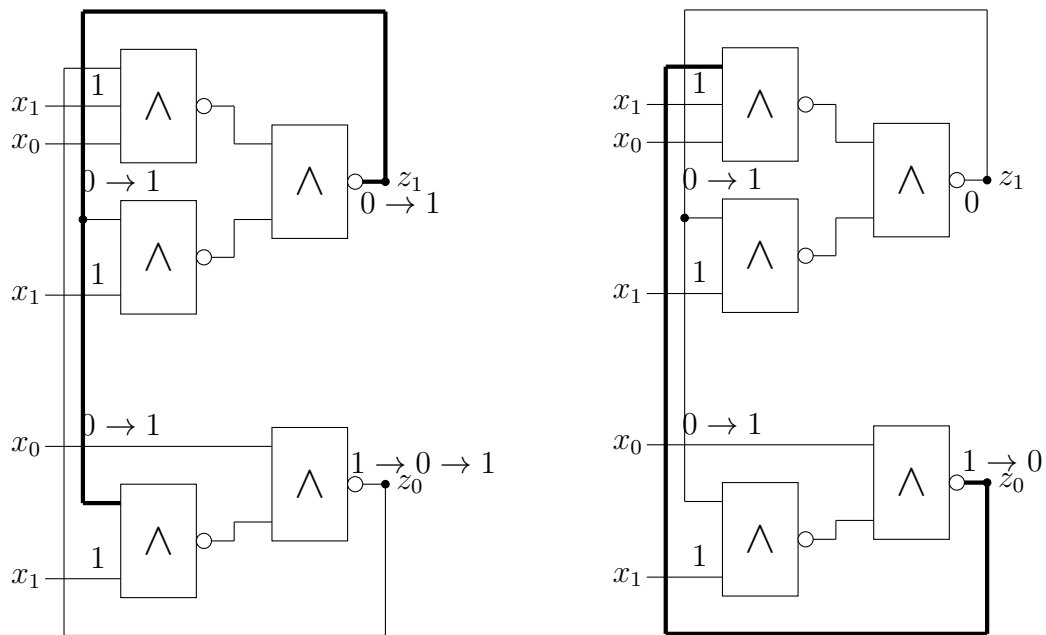


Abbildung 2.28: Race  $z_1, z_0 = 11$  bzw.  $00$

### 2.5.2.1 Allgemeine Formeln zu Races

Gegeben ist eine asynchrone Rückkopplungsschaltung mit  $n$  Zuständen und damit  $n$  Rückkopplungen im Single-Rail-Design, siehe Abbildung 2.30. Daraus wird die allgemeine Wettlaufformel in Gleichung 2.44 abgeleitet.

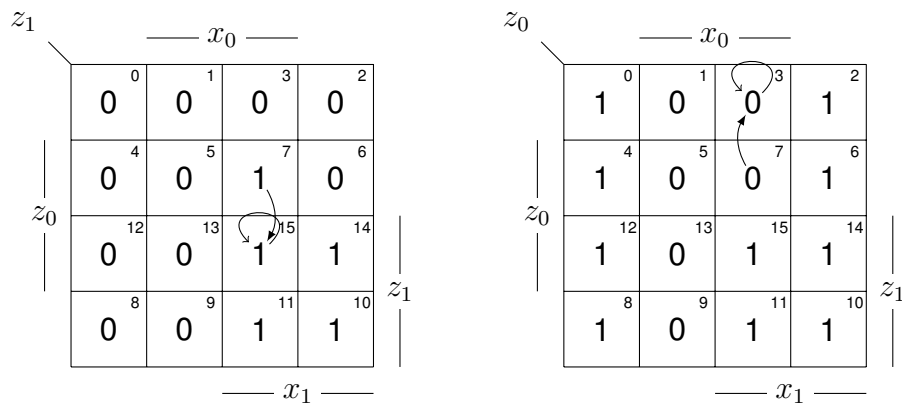


Abbildung 2.29: KV-Diagramme für  $z_1$  und  $z_0$

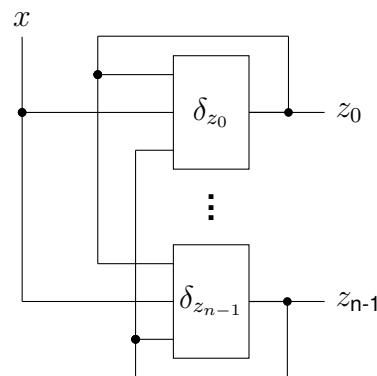


Abbildung 2.30: Allgemeine Asynchrone Feedback-Schaltung

Ein Race kann nur entstehen, wenn zwei Zustände gleichzeitig wechseln und der nachfolgende Zustand vom Ausgang des Races abhängt (d. h. in der realen Schaltung beeinflussen die Rückkopplungsleitungen das Ergebnis des Races).

Eine gleichzeitige Änderung der beteiligten  $z$ -Variablen führt zu einer Überlagerung der betroffenen Übergänge.

Zur Berechnung der doppelten Übergänge betrachten wir zunächst einen Übergang  $\tilde{\Delta}$  von 0 auf 1 in einer Rückkopplungsschaltung  $\delta_z$ . Die 1-Werte  $\Delta_z$  sind gegeben, aus denen wir nun die Startpunkte bestimmen, indem wir den Testmodus  $\bar{Z}$  setzen.

Gleichung 2.39 beschreibt dieses Verfahren in Positive Logik (PL).

$$\tilde{\Delta}_z = \bar{Z} \wedge \Delta_z(x) = \Delta_z(x)|_{\bar{Z}} \quad (2.39)$$

Gleichung 2.40 zeigt alle Startpunkte  $\tilde{\delta}_{z_0}$  der Übergänge von  $\delta_{z_0}$  von 0 auf 1 sowie von

$\bar{\delta}_{z_0}$  von 1 auf 0 im Dual-Rail-Design und die zugehörigen Testmodi.

$$\tilde{\delta}_{z_0} = (\tilde{\Delta}_{z_0}, \tilde{\bar{\Delta}}_{z_0}) = (\bar{Z}_0 \wedge \Delta_{z_0}, Z_0 \wedge \bar{\Delta}_{z_0}) = (\Delta_z(x)|_{\bar{z}}, \bar{\Delta}_{\bar{z}}(x)|_z) \quad (2.40)$$

Die Überlagerung zweier Übergänge führt zu den doppelten Übergängen, wie in Gleichung 2.41 dargestellt.

$$\begin{aligned} \tilde{\Delta}_{z_1 z_0} &= \bar{Z}_1 \bar{Z}_0 \wedge (\Delta_{z_1} \Delta_{z_0}) \\ \tilde{\bar{\Delta}}_{z_1 \bar{z}_0} &= Z_1 Z_0 \wedge (\bar{\Delta}_{z_1} \bar{\Delta}_{z_0}) \end{aligned} \quad (2.41)$$

Die allgemeine Formel für alle 0-zu-1-Doppelübergänge von  $n$   $z$ -Variablen lautet:

$$\tilde{\Delta} = \bigvee_i (\bar{Z}_i \wedge \Delta_{z_i} \bigwedge_j \bar{Z}_j \wedge \Delta_{z_j}) \quad (2.42)$$

mit  $i \in \{0, \dots, n-1\}$  und  $j \in \{i+1, \dots, n-1\}$ . Die zu erfüllende Randbedingung ist die Abhängigkeit von einer bestimmten Rückkopplungsleitung. Die Bestimmung, ob eine einzelne  $z$ -Variable von sich selbst oder einer anderen  $z$ -Variable abhängt, wird durch die partielle Ableitung in Gleichung 2.43 ausgedrückt.

$$\frac{\partial \delta_{z_i}}{\partial z_k} = \delta_{z_i}(z_k) \approx \delta_{z_i}(\bar{z}_k) \quad (2.43)$$

wobei  $i \in \{0, \dots, n-1\}$  und  $k \in \{0, \dots, n-1\}$  gilt. Die allgemeine Formel für die fatalen Races  $c_{z_i z_j}$  der Zustände  $z_i$  und  $z_j$  ist in Gleichung 2.44 gegeben, während die Gleichung für nicht-kritische Races  $c_{z_i}$  für den Zustand  $z_i$  in Gleichung 2.45 dargestellt ist.

$$c_{z_i z_j} = \left( \frac{\partial \delta_{z_i}}{\partial z_j} \wedge \frac{\partial \delta_{z_j}}{\partial z_i} \wedge \tilde{\delta}_{z_i z_j} \right) \exists_{z_i z_j} \quad (2.44)$$

$$c_{z_i} = \left( \frac{\partial \delta_{z_i}}{\partial z_j} \wedge \overline{\frac{\partial \delta_{z_j}}{\partial z_i}} \wedge \tilde{\delta}_{z_i z_j} \right) \exists_{z_i z_j} \quad (2.45)$$

### 2.5.2.2 Vermeidung von Races

Um Races zu vermeiden, werden in [10] die folgenden Möglichkeiten definiert:

#### **Definition 10** *Vermeidung von Races*

- Vermeidung von Races durch Taktbetrieb.
- Vermeidung von Races durch eine Binärcodierung mit der Hamming-Distanz  $D=1$ : Jeder Zustandswechsel muss durch Änderung von nur einer Zustandsvariablen geschehen.
- Vermeidung von Races durch Anwendung einer Standardkodierung: Jedem Zustand wird genau eine Zustandsvariable zugesprochen, sodass die Stelligkeit des Zustandsvektors der Anzahl an Zuständen entspricht (z.B. One-hot-Kodierung).
- Vermeidung von Races durch Anwendung einer Übergangstrennkodierung: Bei potentiellen Race-Zuständen umkodieren, sodass die zwei konkurrierenden Zustände disjunkt zueinander sind.

Anmerkung zur Standardkodierung: Es muss zuerst der Zustandswechsel der Variablen geschehen, welche neu auf 1 gesetzt wird, sodass kurzzeitig zwei logische 1-er anliegen und dann die andere Zustandsvariable auf 0 gesetzt werden, da jeder Zustandswechsel den Nullpunkt durchlaufen könnte und somit mit gleichem Eingang ein anderer Zustand auftreten könnte. Dies kann durch schaltungstechnische Maßnahmen verhindert werden.

### 2.5.3 Glitches

In digitalen Systemen sind Glitches unerwünschte, vorübergehende Spannungsspitzen oder -schwankungen, die auftreten [33]. Funktions hazards werden in der Literatur oft auch als Glitch bezeichnet, in dieser Thesis wird der Begriff Glitch aber für unerwünschte Störsignale genutzt, die keinen definierten Ursprung haben. Unter diese Kategorie fallen auch die Soft-Errors, Soft-Transients oder Single Event Upset (SEU)s. Es handelt sich dabei um kurzlebige Impulse, die sich in einer Schaltung ausbreiten können und möglicherweise unerwartetes Verhalten oder fehlerhafte Berechnungen verursachen. Glitches entstehen durch Zeitverzögerungen in der Schaltung, alpha-Partikel oder ionisierende Strahlungen [34]. Handelt es sich bei einem Störsignal um einen andauernden Fehler, z.B. wenn dadurch ein falsches Bit gespeichert wird, spricht man von

einem SEU. Single-Event-Latchups können das Bauteil sogar zerstören, werden in dieser Arbeit aber nicht thematisiert. Obwohl sie in der Regel nur von kurzer Dauer sind, können Glitches erhebliche Auswirkungen auf die Systemleistung haben, insbesondere bei Hochgeschwindigkeits- und Hochleistungsschaltungen wie FPGAs, Prozessoren und anderen integrierten Schaltungen.

### **2.5.3.1 Auswirkungen von Glitches und SEUs**

Glitches und SEUs wirken sich auf verschiedene Weise in Schaltungen aus, diese werden im Folgenden kurz erläutert.

#### **Leistungsverbrauch**

Glitches erhöhen die dynamische Leistungsaufnahme einer Schaltung, da die Impulse oft zu unnötigen Übergängen führen, verursachen sie zusätzliche Schaltaktivitäten, die die Gesamtverlustleistung in digitalen Schaltungen erhöhen [33]. Dieses Problem ist vor allem bei Systemen mit geringem Stromverbrauch, wie z. B. mobilen und eingebetteten Geräten, von Bedeutung, bei denen die Energieeffizienz ein wichtiger Gesichtspunkt bei der Entwicklung ist.

#### **Datenfälschungen**

In synchronen Systemen können sich SEUs auf getaktete Speicherelemente wie Flip-Flops ausbreiten, wenn sie nahe an einer Taktflanke auftreten. In diesem Fall können SEUs dazu führen, dass falsche Daten zwischengespeichert werden, was zu Datenverfälschung oder falschen Berechnungen führt [35]. Darüber hinaus können Glitches, die sich über kritische Pfade ausbreiten, zu Verletzungen der Setup- oder Hold-Zeit führen und das Zeitverhalten des Systems stören.

#### **Funktionale Ausfälle**

In komplexen Systemen wie Prozessoren oder FPGAs können Glitches funktionale Fehler auslösen, z. B. falsche Verzweigungen, fehlerhafte Speicherzugriffe oder unerwartete Rücksetzungen. Da die Schaltkreise immer kompakter werden und mit höheren Geschwindigkeiten arbeiten, können sich selbst kleine Störungen über mehrere Stufen der Logik ausbreiten und zu systemweiten Fehlfunktionen führen.

## 2.6 Asynchroner Entwurf

Hauptziel dieser Arbeit ist es, den Entwurfsprozess asynchroner Schaltungen im FPGA zu zeigen. Hierfür soll im folgenden Kapitel ein Einblick in die asynchrone digitale Welt erfolgen. Ein umfassender Überblick über das asynchrone Design findet sich bei Sparso [36], [37]. [31] ordnet in seiner Dissertation asynchrone Schaltungen hierarchisch an und unterscheidet dabei zwischen delay-insensitive, speed-independent und self-timed Schaltungen.

Delay-insensitive Schaltungen sind so konzipiert, dass sie korrekt funktionieren, unabhängig von Verzögerungen in Gattern und Leitungen [38]. Sie gelten als die robusteste Klasse asynchroner Schaltungen, da sie keine Annahmen über Verzögerungen treffen. Allerdings sind sie in ihrer praktischen Anwendung begrenzt, da nur sehr einfache Schaltungen vollständig delay-insensitive realisierbar sind.

Speed-independent Schaltungen [39] hingegen berücksichtigen Verzögerungen in Gattern, gehen jedoch davon aus, dass Leitungsverzögerungen vernachlässigbar sind. Diese Annahme ermöglicht eine größere Vielfalt an Schaltungsdesigns, erfordert jedoch sorgfältige Gestaltung, um sicherzustellen, dass Leitungsverzögerungen die Funktion nicht beeinträchtigen.

Self-timed Schaltungen [40], die auch als bounded-delay oder quasi-delay-insensitive Schaltungen bekannt sind, basieren auf der Annahme, dass sowohl Gatter- als auch Leitungsverzögerungen innerhalb definierter Grenzen liegen. Zur Synchronisation setzen sie häufig auf Handshake-Protokolle, wodurch eine ausgewogene Kombination aus Robustheit und Effizienz erreicht wird. Diese gezielten Timing-Annahmen erleichtern die Implementierung und machen sie zu einer attraktiven Alternative in hochoptimierten digitalen Schaltungsdesigns. Self-timed Schaltungen können noch weiter unterteilt werden. Zum einen gibt es Schaltungen, die den Takt von einer extra Schaltung von außen vorgegeben bekommen im Sinne eines echten Taktes, es gibt aber auch Schaltungen, die durch die Propagation der Schaltung an sich ihren eigenen Takt innerhalb erzeugen. Werden verschiedene self-timed Schaltungen kombiniert entsteht eine GALS-Schaltung. GALS haben keinen globalen Takt, sondern verschieden verteilte Einzeltakte, die z.B. durch Handshaking entstehen können. Alle Schaltungsarten werden dabei vom Begriff asynchron, also alles außer synchron, umschlossen. Für diese Thesis werden delay-insensitive Schaltungen und self-timed Schaltungen betrachtet und als Beispiel zur Anwendung wird ein RISC-V Prozessor als GALS-System realisiert. Ein Vererbungsdiagramm zu den verwendeten Schaltungen ist in Abbildung 2.31

gegeben. GALS können in ihren Strukturen self-timed und delay-insensitive Schaltungen

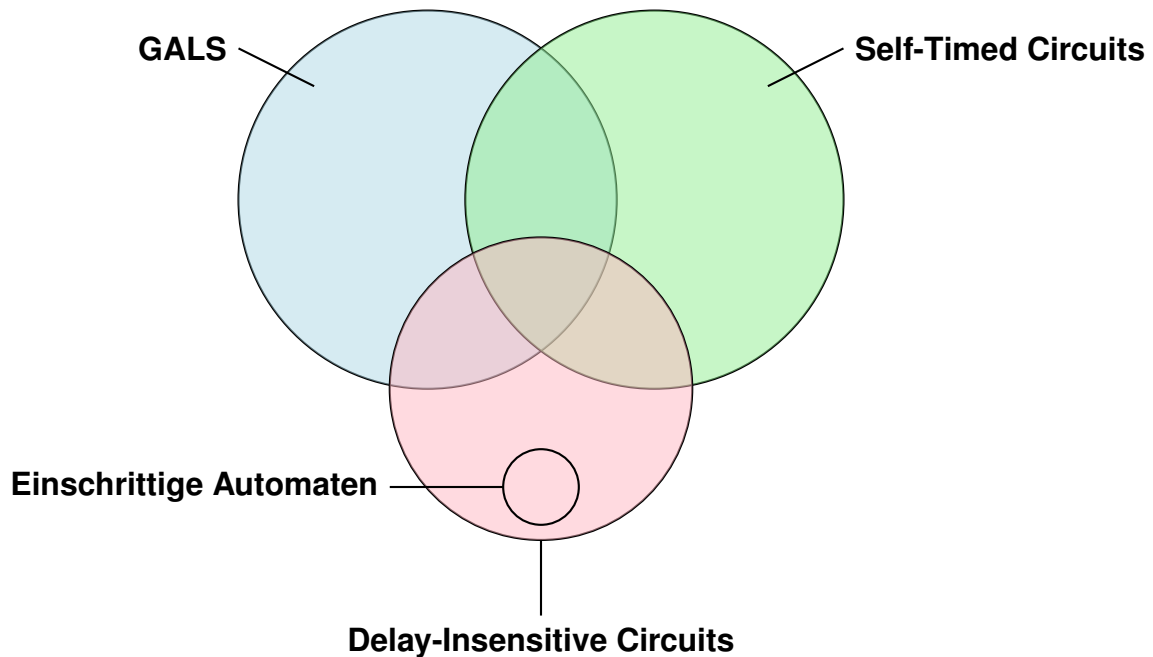


Abbildung 2.31: Venn-Diagramm der behandelten Asynchronen Schaltungen

gen haben, sodass diese in diesem Bereich überlappen. Eine self-timed Schaltung kann auch einen Anteil an delay-insensitiven Schaltungen haben, komplett davon abgekoppelt sind aber die einschrittigen Automaten, da diese taktlos, also auch ohne Handshaking, mit sehr strikten Implementierungsvorgaben realisiert werden müssen. Die Grundlagen, die zum Verständnis des asynchronen Entwurfs beitragen werden im Folgenden kurz erläutert.

### 2.6.1 Strukturtreue Modellierung

Unter strukturtreuer Modellierung wird die Übereinstimmung der formal hergeleiteten Funktion mit der real erzeugten Funktion verstanden. Für die funktionale Sicherheit von sicherheitskritischen Schaltungen und Systemen muss die der realen Struktur zugehörige Funktion strukturtreu modelliert werden; es muss also gewährleistet sein, dass die formal hergeleitete und modellierte Funktion mit der durch die reale Struktur erzeugten Funktion konsistent übereinstimmt. Umgekehrt ist es sicherheitstechnisch fatal, wenn die modellierte Funktion ein Verhalten aufweist, welches sich von der realisierten Funktion unterscheidet [41].

Die realisierte Struktur soll weiterführend strukturtreu modelliert werden. Jede Struktur stellt genau eine Realisierung für ihre Funktion dar. Für eine Struktur (Realität) gibt

es eine eindeutige Funktion (Verhalten), eine Funktion kann aber mit beliebig vielen Strukturen realisiert werden. Jeder Pin im Modell muss in der Realität (Struktur) vorkommen. Die eineindeutige Übereinstimmung ist gegeben, wenn die der spezifizierten Funktionalität zu Grunde liegende Struktur die selbe Funktionalität realisiert. Eineindeutige Modellierung bedeutet:

- aus der Sicht der realen Struktur, dass jedes Symbol im Modell als Pin in der Struktur vorkommt und
- aus der Sicht des modellierten Verhaltens (Modell), dass jede getroffene Aussage (Formel, Tabelle, Skizze) wahr ist, sprich in der Realität vorkommt. [42]

In der Thesis wird die Konvention beibehalten, dass Verhalten (z.B. Signale) mit Kleinbuchstaben (KBS) und Realitäten (z.B. Pins) mit GBS bezeichnet werden. Genauso werden binäre Variablen mit KBS und unäre Größen mit GBS dargestellt.

### 2.6.2 Dual-Rail-Logik

Dual-Rail-Logik stellt binäre Systeme durch zwei Rails dar, ein Rail für die 1-er, das andere für die 0-er. In dieser Logik gibt es zwei valide Signalkombinationen [01, 10], die die logischen Werte 0 und 1 repräsentieren [43], sowie invalide Kombinationen [00, 11]. Kommt es durch ungewollte Situationen zu den Wertekombinationen [00, 11] soll dieser Fehler abgefangen werden, indem keine Wertänderung passiert.

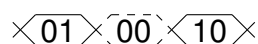


Abbildung 2.32: Umschaltvorgang bei der Dual-Rail-Logik

Typischerweise erfolgt das Switchen lediglich über die Wertekombination [00], siehe Abbildung 2.32, sodass es sich um monolithisches Switchen handelt. Dadurch, dass es nur zu einer Änderung kommt, wenn das 1-Rail und das 0-Rail disjunkt zueinander sind, können Hazards durch ungewollte Verzögerungen, und dadurch entstehendem ungleichen Umschalten von Variablen, abgefangen werden. Das Handshaking beider Rails wird in der Literatur üblicherweise durch ein Muller-C-Element vorgenommen [44].

### 2.6.3 Muller-C-Element

Das Muller-C-Element beschreibt eine binäre Logikschaltung, die häufig in asynchronen Schaltungen verwendet wird [44]. Es hat zwei Eingänge (A und B) und einen Ausgang (Q), siehe Abbildung 2.33. Die Ausgabe des Muller-C-Elements hängt von

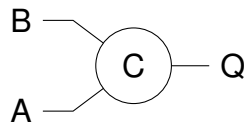


Abbildung 2.33: Muller-C-Symbol

den aktuellen Eingangswerten und dem vorherigen Zustand des Ausgangs ab. Das Muller-C-Element kann als ein Speicher betrachtet werden, der den letzten Zustand der Eingänge merkt, falls diese invalide sind. Wenn beide Eingänge 0 sind, wird der Ausgang 0 sein. Wenn beide Eingänge 1 sind, wird der Ausgang 1 sein. In allen anderen Fällen behält der Ausgang seinen vorherigen Zustand bei. Die fWertetabelle des Muller-C-Elements ist in Tabelle 2.6 gegeben. Das Muller-C-Element beschreibt, wie

Tabelle 2.6: Wahrheitstabelle des Muller-C

A	B	Q
0	0	0
0	1	Q
1	0	Q
1	1	1

bereits erwähnt, die Funktion, es kann in unterschiedlichsten Strukturen vorliegen. Eine Struktur, welche die Funktion des Muller-C realisiert, ist der RS-Buffer, welcher in der Literatur als semi-statische Implementierung angegeben wird. Weitere Implementierungen sind u.A. statische Realisierungen oder die Realisierung der Funktion auf Gatter-Level.

#### 2.6.3.1 Probleme von Muller-C-Realisierungen

Das Muller-C-Element beschreibt die Funktion, kann aber durch verschiedene Strukturen realisiert werden. Werden einige Realisierungen genauer betrachtet, wie z.B. das Muller-C aus Logikgattern, siehe Abbildung 2.34, kommen Constraints dazu, die zu beachten sind. Es ist zu erkennen, dass die Setup- und Hold-Zeit durch die äußere Rückkopplung sehr hoch ist, da eine Transition durch zwei Stufen und zusätzlich durch

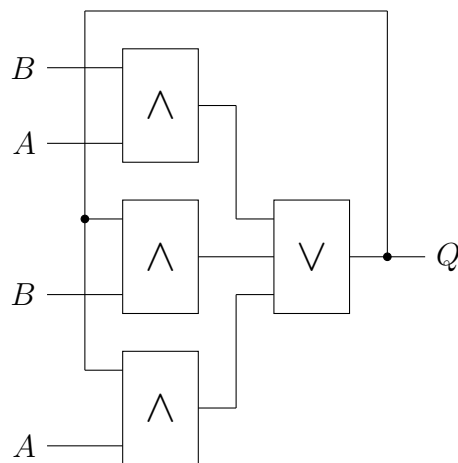


Abbildung 2.34: Signalflussplan des Muller-C

die Verzögerungszeit der Rückkopplung hindurch propagieren muss, d.h. das Signal muss so lange stabil anliegen, bis es sich selbst aufrechterhalten kann. Im Folgenden werden wir eine Struktur des Muller-C genauer betrachten und als RS-Buffer bezeichnen, wenn diese Struktur gemeint ist.

#### 2.6.4 RS-Buffer

Der RS-Buffer kann als Vertreter des Muller-C bezeichnet werden, da er die Realisierung mit der geringsten Fläche und geringsten Fehleranfälligkeit ist. Er ist selbstsperrend und weist eine Hamming-Distanz von zwei für valide Datensignale auf. Die Transistorschaltung ist in Abbildung 2.14 gegeben, die zugehörige Tabelle liegt in Tabelle 2.7 vor. B steht dabei für den Ausgangswert des RS-Buffers. Die drei Funktionen

R	S	B	Kommentar
0	0	B	Hold
0	1	1	Set
1	0	0	Reset
1	1	B	Hold

Tabelle 2.7: Wahrheitstabelle des RS-Buffers

sind das Setzen der 1, das Rücksetzen zur logischen 0 und der Hold-Betrieb, wenn ein unplausibles Signal vorkommt. Dadurch, dass es sich bei den Eingängen [00] und [11] um eine Abhängigkeit vom alten Zustand des RS-Buffer handelt, liegt hier eine sequentielle Struktur vor. Hierfür wird zunächst das KV-Diagramm, siehe Abbildung 2.35, betrachtet. Da der RS-Buffer lediglich eine Rückkopplung besitzt, besteht keine Gefahr

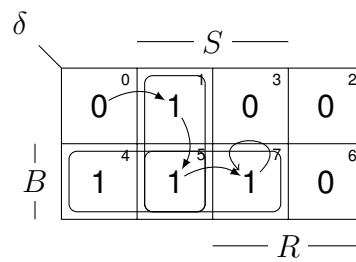


Abbildung 2.35: KV-Diagramm des RS-Buffers

von Races. Das KV-Diagramm lässt zwar Funktionshazards vermuten, jedoch zeigt die Funktion durch die zwei Transitionspunkte und den sonst zustandsstabilen Anteilen ein stabiles Verhalten auf. Beispielhaft soll kurz anhand des KV-Diagramms gezeigt werden, wieso es zu keinem Funktionshazard kommt. Man könnte z.B. vermuten, dass ein Funktionshazard von  $X_0$  nach  $X_3$  auftreten würde. Da der Zustand aber eine Transition durchläuft und sich dann selbst bestätigt, ist ein Funktionshazard ausgeschlossen. Analog gilt das Gleiche für den Fall, dass sich R zuerst ändert. Ein weiterer Fall, der zu einem theoretisch möglichen Funktionshazard führen könnte, ist das Schalten von  $[00]$  nach  $[11]$ . Da so aber beim RS-Buffer nicht geschaltet wird, sondern lediglich von validen Signalen zu validen Signalen, ist auch dieser theoretische Funktionshazard kein Problem.

Die Formel in AA für den RS-Buffer ist in Gleichung 2.46 gegeben.

$$B = S\bar{R} \vee SB \vee \bar{R}B \quad (2.46)$$

### 2.6.5 Asynchrones stabiles Automatenesign durch reduzierte Übertragungsfunktionen

Der folgende Abschnitt bezieht sich auf die Veröffentlichungen von Özgül et al. [45]–[47]. Der zentrale Gedanke des stabilen Entwurfs besteht darin, sowohl eine stabile ZÜF als auch eine stabile Ausgabefunktion zu erhalten, sodass deren Vereinigung einen stabilen Automaten ergibt. Zur Stabilisierung der ZÜF wird diese zunächst eingehender untersucht.

#### 2.6.5.1 Zustandsstabile Zustandsüberföhrungsfunktion

Ein eindimensionaler Automat besitzt genau eine z-Variable  $z$  und damit genau eine z-Gleichung. Diese wird nun näher betrachtet. Es wird ein spezielles Augenmerk auf

den Zusammenhang zwischen dem alten und neuen Zustand gelegt. Dieser kann zwei Werte besitzen  $z$  und  $\bar{z}$  wodurch  $2^2$  verschiedene Kombinationen entstehen. Diese sind beispielhaft in Abbildung 2.36 zu sehen. Wird das KV-Diagramm so gefaltet, dass

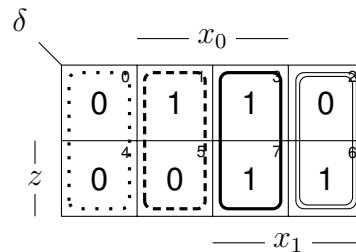


Abbildung 2.36: Vier Teile einer Zustandsüberföhrungsfunktion

die  $z$ -Dimension in den einzelnen Zellen steht, ergibt sich das zusammengefasste KV-Diagramm siehe Abbildung 2.37. Zu sehen ist, dass es einen vom letzten Zustand

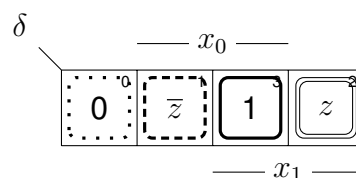


Abbildung 2.37: Gefaltetes KV-Diagramm

unabhängigen kombinatorischen 0-Anteil gibt sowie einen kombinatorischen 1-Anteil und zwei Teile, die vom letzten Zustand abhängig sind. Zum einen die Identität (zustandsstabiler Anteil) und zum anderen die Negation (zustandsinstabiler Anteil). Die vier Teile nach obigem Beispiel sollen nun in Tabelle 2.8 nochmals verdeutlicht werden. Der Eingangsvektor  $\underline{x}_i$  steht dabei für den konstanten Eingang der seiner Zahl zugeordnet ist, d.h.  $\underline{x}_0 = \bar{x}_1 \bar{x}_0$ . Für die Eingangsbelegung  $\underline{x}_0$  ist der Zustandsüberföhr-

$\#\underline{x}$	$z\delta(\underline{x}_i)$	$\bar{z}\delta(\underline{x}_i)$	$\delta$	Kommentar
$\underline{x}_0$	0	0	0	transiente 0
$\underline{x}_1$	0	1	$\bar{z}$	zustandsinstabiler Anteil
$\underline{x}_2$	1	0	$z$	zustandsstabiler Anteil
$\underline{x}_3$	1	1	1	transiente 1

Tabelle 2.8: Tabellarische Darstellung der vier Teile

rungsfunktionswert unabhängig vom Wert des alten Zustand auf der logischen 0. Es handelt sich also um einen Teil der zu einer Transition führen kann, genauer gesagt der zu einer 0-Transition führen kann. Der Eingangsvektor  $\underline{x}_1$  ist hingegen abhängig vom

alten Zustand, genauer gesagt die Negation des alten Zustands. Es handelt sich um einen zustandsinstabilen Anteil, da die ZÜF bei gleichbleibender Eingangsbelegung zu oszillieren beginnt, da jeweils der alte Zustand negiert wird. Der nächste Teil der Funktion mit der Eingangsbelegung  $x_2$  ist auch vom alten Zustand abhängig. Hier ist er jedoch eine Identität des alten Zustands und somit der zustandsstabile Anteil. Die Eingangsbelegung  $x_3$  zeigt den transienten 1-Anteil. Dieser ist wie die 0-Transition unabhängig vom alten Zustand und kann zu einer 1-Transition führen.

Da das asynchrone Automatendesign (funktions-)stabil sein soll, wird der zustandsinstabile, in einer Oszillation endende, Anteil der ZÜF für den Entwurf verboten. Übrig bleiben noch drei Teile der ZÜF: die 1-Transition, die 0-Transition und der zustandsstabile Anteil. Die ZÜF kann also durch eine ternäre Kodierung dargestellt werden.

Die einzelnen Teile der ZÜF lassen sich formell beschreiben. Hierfür müssen zunächst einige Ausdrücke definiert werden:

**Definition 11** *Schreibweise von  $\delta$  in Bezug auf  $z$*

$${}_z\delta(z, x) = \delta(z = 1, x) \quad (2.47)$$

$${}_{\bar{z}}\delta(z, x) = \delta(z = 0, x) \quad (2.48)$$

$${}_z\bar{\delta}(z, x) = \bar{\delta}(z = 1, x) \quad (2.49)$$

$${}_{\bar{z}}\bar{\delta}(z, x) = \bar{\delta}(z = 0, x) \quad (2.50)$$

Die einzelnen Teile der ZÜF können nun formell beschrieben werden durch:

$$\text{transiente 1 : } \underline{\delta}_z = {}_z\delta_z \wedge {}_{\bar{z}}\delta_z \quad (2.51)$$

$$\text{transiente 0 : } \bar{\delta}_{\bar{z}} = {}_z\bar{\delta}_{\bar{z}} \wedge {}_{\bar{z}}\bar{\delta}_{\bar{z}} \quad (2.52)$$

$$\text{zustandsstabiler Anteil : } {}_z\delta_z \wedge {}_{\bar{z}}\bar{\delta}_{\bar{z}} \quad (2.53)$$

$$\text{zustandsinstabiler Anteil : } {}_z\bar{\delta}_{\bar{z}} \wedge {}_{\bar{z}}\delta_z \quad (2.54)$$

### 2.6.5.2 Entwurf der stabilen Ausgabefunktion

Da es sich bei der Ausgabefunktion um den kombinatorischen Teil des Automaten handelt, muss die Ausgabefunktion lediglich hazardfrei entworfen werden. Dieses gelingt durch den RS-Buffer und entsprechendes Entwerfen der Dual-Rail-Logik. Schaltungsblöcke werden durch ihre Resolvente überlappend entworfen, um Strukturhazards in zweistufigen Schaltungen zu vermeiden. Der stabile Automat mit der stabilen ZÜF  $\underline{\delta}$

und der stabilen Ausgabefunktion  $\underline{\mu}$  ist in Abbildung 2.38 in allgemeiner Form dargestellt.

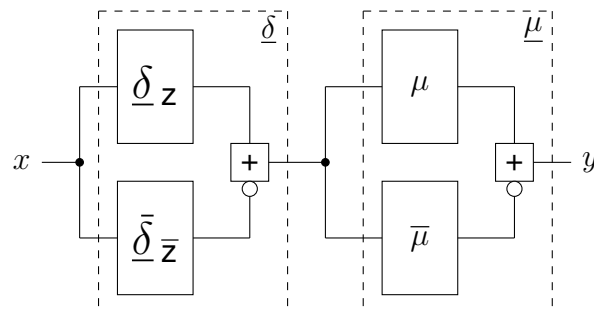


Abbildung 2.38: Stabiler Moore-Automat

### 2.6.5.3 Vor- und Nachteile des stabilen Automatenentwurfstils

In diesem Abschnitt wird kurz auf die Vor- und Nachteile dieses Entwurfstils eingegangen. Wird das partielle RS-Latch mit dem partiellen RS-Latch aus [23] verglichen, ist eindeutig zu erkennen, dass diese Konfiguration im Vergleich zu einer geringeren Komplexität führt, i.e. weniger Schaltungsaufwand benötigt. Dies gilt in dieser Konfiguration. Sollten sich jedoch die 1- und 0-Anteile besser kombinieren lassen, ist dies nicht zwingend der Fall. Es ist also nicht oBdA zu sagen, dass besagter partieller Automatenentwurfstil zu simpleren Schaltungen führt. Ein Nachteil hingegen ist, dass durch die Parteilheit die Hamming-Distanz von 2 verloren geht und somit keine Glitch-Freiheit mehr besteht. Es ist möglich die \*-Belegungen mit einem Hold zu kodieren, wodurch dann wieder die Schaltung aus [23] entsteht.

### 2.6.6 Handshake-Protokolle

Ein asynchrones Handshake-Protokoll stellt eine Kommunikationsvereinbarung zwischen zwei oder mehr Entitäten dar, die den Datenaustausch ohne die Notwendigkeit einer gemeinsamen Taktfrequenz ermöglicht [48]. Im Gegensatz zu synchronen Protokollen, die sich auf das Timing eines gemeinsamen Takts stützen, um die Kommunikation zu regeln, verwenden asynchrone Handshake-Protokolle ein Paar von Signalen zur Steuerung der Datenübertragung. Das erste Signal dient zur Initiierung der Datenübertragung (*req*), während das nachfolgende Signal zur Bestätigung des erfolgreichen Abschlusses der Datenübertragung (*ack*) genutzt wird.

### 2.6.7 4-Phasen-Handshake-Protokoll

Zunächst wird das Vier-Phasen-Protokoll, wie in [36] beschrieben, erläutert, siehe Abbildung 2.39.

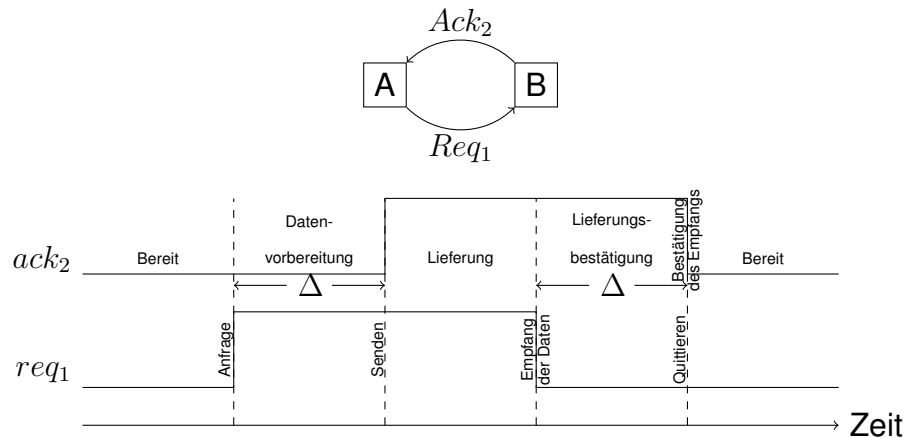


Abbildung 2.39: 4-Phasen-Handshake-Protokoll

Der Prozess beginnt damit, dass  $B$  den Kommunikationskanal öffnet und signalisiert, dass es bereit ist, eine Anfrage zu empfangen, indem es  $ack_2$  auf low setzt. Anschließend möchte  $A$  eine Datenübertragung initiieren und signalisiert dies, indem es  $req_1$  auf high setzt. Daraufhin verarbeitet  $B$  die Anfrage und bereitet die entsprechenden Daten vor.

Sobald  $B$  bereit ist, die Daten zu senden, signalisiert es dies, indem es  $ack_2$  auf high setzt.  $A$  empfängt die Daten und bestätigt den Erhalt, indem es  $req_1$  wieder auf low setzt. Schließlich erkennt  $B$  die Bestätigung von  $A$  und setzt  $ack_2$  wieder auf low, womit der Handshake-Zyklus abgeschlossen ist und eine neue Übertragung beginnen kann.

### 2.6.8 2-Phasen-Handshake-Protokoll

Die Informationen auf den Request- und Acknowledge-Pins werden jetzt als Signalübergänge auf den Drähten kodiert, und es gibt keinen Unterschied mehr zwischen einem 0 auf 1 und einem 1 auf 0 Übergang, sie stellen beide ein Signalereignis dar, siehe Abbildung 2.40.

Der Prozess beginnt damit, dass  $B$  den Kommunikationskanal öffnet und signalisiert, dass es bereit ist, eine Anfrage zu empfangen, indem es seinen Wert hält, also kein Signalübergang vorliegt. Sobald  $A$  eine Datenübertragung starten möchte, setzt es  $req_1$

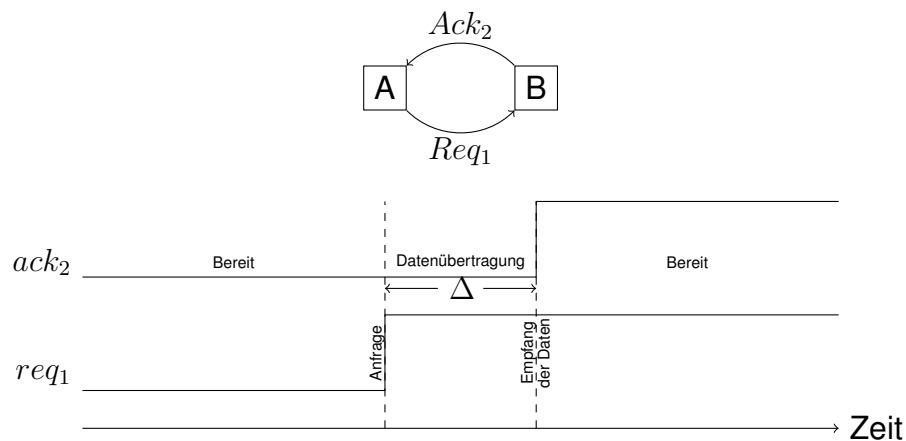


Abbildung 2.40: 2-Phasen-Handshake-Protokoll

von low auf high oder high auf low. Daraufhin verarbeitet  $B$  die Anfrage und bestätigt den Empfang sowie die Verarbeitung der Daten, indem es  $ack_2$  umschaltet (von low auf high oder von high auf low). Sobald  $A$  die Bestätigung durch  $B$  erkennt, kann es die nächste Übertragung starten, indem es  $req_1$  erneut umschaltet. Damit beginnt ein neuer Handshake-Zyklus. Es handelt sich also um ein non-return-to-zero-Protokoll.

## 2.7 Einführung in die Krohn-Rhodes-Theorie

Der folgende Abschnitt basiert größtenteils auf den Arbeiten von [49], [50]. Die Krohn-Rhodes-Theorie ist ein fundamentaler Satz in der Darstellung von Automaten und wurde von Kenneth Krohn und John Rhodes in den 1960er Jahren entwickelt. Sie besagt, dass jeder beliebige endlichen Automaten in eine Kaskade einfacherer Bausteine zerlegt werden kann. Diese Bausteine bestehen aus sogenannten Reset-Automaten und Permutationsgruppen und bieten eine methodische Grundlage für das Verständnis und die Analyse komplexer Systeme. Der Satz kann direkt für die Untersuchung und Implementierung endlicher Automaten genutzt werden, da jeder Automat als Halbgruppe gesehen werden kann. In dieser Thesis soll die Krohn-Rhodes-Theorie ausgenutzt werden, um endliche Automaten in eine Kaskade zu zerlegen, um asynchrone Dominiologik zu realisieren.

### Definitionen und Grundbegriffe

Die formale Beschreibung der Krohn-Rhodes-Theorie erfordert einige grundlegende Definitionen im Folgenden sollen die Grundbausteine, also die einfachen Automaten,

die für die Krohn-Rhodes-Theorie wichtig sind, erläutert werden.

### 2.7.0.1 Endlicher Automat

Zunächst wird der endliche Automat  $A$  aus Gleichung 2.23 erweitert um den Startzustand  $Z_0$  und die Menge  $F \subseteq Z$ . Die Krohn-Rhodes-Theorie besagt nun, dass jeder Automat  $A$  in Unterstrukturen aus zwei verschiedene Strukturen besteht, den sogenannten Primes und Units. Eine Gruppe  $S$  wird als Prime bezeichnet, wenn sie eine nichttriviale einfache Gruppe ist. Das bedeutet:

#### Primes

- $S$  ist eine Gruppe mit einer eindeutigen Multiplikationsstruktur (Permutation). In einer Gruppe hat jedes Element ein inverses Element (für jede Transition gibt es eine Rücktransition).
- $S$  besitzt keine echten Normalteiler außer der trivialen Untergruppe  $\{e\}$  und sich selbst. Das bedeutet, dieser Automat besitzen keine zerlegbaren Unterstrukturen, außer sich selbst und dem trivialen Zustand.

Mathematisch kann die Menge  $P$  aller Primes definiert werden als:

$$\text{PRIMES} = \{P \mid P \text{ ist eine nichttriviale einfache Gruppe}\}. \quad (2.55)$$

Primes tragen zur Permutationsstruktur der Zerlegung bei.

Falls  $S$  keine einfache Gruppe ist, dann ist es eine der drei möglichen Teiler einer bestimmten dreielementigen Halbgruppe  $U_3$ . Diese Strukturen werden als Units bezeichnet und sind fundamentale Bausteine für nicht-permutative Eigenschaften eines Automaten.

Die Halbgruppe  $U_3$  ist definiert als eine dreielementige Halbgruppe. Sie enthält drei Elemente:

$$U_3 = \{u_2, u_1, u_0\} \quad (2.56)$$

mit bestimmten Multiplikationsgesetzen.

Die möglichen **Units** sind genau die Unterstrukturen von  $U_3$ , nämlich:

- $U_0 = \{1\}$ , die triviale Halbgruppe mit nur einem einzigen Element. Dies kann als Automat mit nur einem einzigen Zustand gesehen werden.
- $U_1 = R_{(\infty)}$ , eine Halbgruppe mit einer stabilen Zustandsstruktur, d.h. eine Art absorbierender Zustand, d. h., einmal erreicht, bleibt der Automat für immer darin.
- $U_2$ , eine zweielementige Halbgruppe, die zwei verschiedene stabile Zustände enthält, z.B. ein FlipFlop, das zwischen zwei Zuständen wechselt.

Mathematisch ist die Menge der Units definiert als:

$$\text{UNITS} = \{S \mid S \text{ teilt } U_3\}. \quad (2.57)$$

Diese Einteilung ist entscheidend für die Zerlegung von Automaten in ihre elementaren algebraischen Bausteine.

Anders ausgedrückt lässt sich jeder Automat durch eine Kombination von Reset-Automaten und Permutationsgruppen modellieren. Es existieren zwei Versionen dieses Satzes:

- **Schwache Form:** Jeder endliche Automat kann als eine Kaskade von Automaten dargestellt werden, die entweder Reset-Automaten oder Automaten mit Permutationsgruppen als Übergänge sind.
- **Starke Form:** Jeder endliche Automat kann als Komposition aus Flip-Flops und der symmetrischen Gruppe  $S_n$  dargestellt werden, wobei  $n$  die Anzahl der Zustände des Automaten ist.

Die Krohn-Rhodes-Theorie stellt dann jede endliche Halbgruppe  $S$  als ein Kranzprodukt von Unterhalbgruppen (subsemigroup) dar. Das Kranzprodukt entspricht der Kaskadierung dieser kleineren Bauteile. Laut der Krohn-Rhodes-Theorie wechseln sich dabei Permutationsgruppen (Primes) und kombinatorische Halbgruppen (Units) ab, beginnend und enden mit den Units. Die minimale Anzahl von Gruppen in einer solchen Darstellung ist die Komplexität  $\theta(S)$  der Halbgruppe  $S$ .

### 2.7.0.2 Serienkomposition

In der Krohn-Rhodes-Theorie wird die Kaskade oder Serienkomposition als Methode verwendet, um Automaten zu kombinieren. Gegeben seien zwei Automaten  $A_1 =$

$(Z_1, X_1, \delta_1, z_{0,1}, F_1)$  und  $A_2 = (Z_2, X_2, \delta_2, z_{0,2}, F_2)$ . Die Serienkomposition  $A = A_1 \cdot A_2$  ergibt einen Automaten, bei dem die Zustände von  $A_1$  und  $A_2$  so kombiniert werden, dass die Ausgabe von  $A_1$  als Eingabe für  $A_2$  dient.

## 2.8 Dominologik

Die DL verdankt ihren Namen den fallenden Dominosteinen - einer Kettenreaktion, bei der ein einziger Impuls eine ganze Reihe von Ereignissen auslöst. In ähnlicher Weise verbindet die Dominologik kaskadierte Strukturen zu einem digitalen System. Im Folgenden soll beginnend mit der Motivation die DL erläutert werden. Anschließend werden SRDL-Gatter sowie deren komplementäre Kombination die DRDL-Gatter vorgestellt. Daraufhin folgt die serielle Komposition dieser Gatter zu einer Kaskade. Die Kaskade wird in folgenden Kapiteln auf dem FPGA implementiert, um asynchrone Schaltungen zu entwerfen. Das Kapitel wurde weitestgehend an [51] angelehnt.

### 2.8.1 Motivation für DL

CMOS-Gatter erfordern viele PMOS-Transistoren mit relativ großen Abmessungen. Die Implementierung eines Gatters mit hohem Fanin (d. h. mit mehr als vier Eingängen) ist schwierig, da entweder der N-Komplex oder der P-Komplex einen großen seriellen Stack erfordert. Daraus entstand der Wunsch, den P-Komplex nicht entwerfen zu müssen, um Fläche sparen zu können, sodass ein immer geöffneter PMOS im Pull-Up (PU) realisiert wird. Die Funktion ist dann in den Pull-Down (PD) geschoben, da dieser entweder leitend ist, wenn die Funktion des PD erfüllt ist und ein kleiner Spannungsabfall passiert oder dieser geschlossen ist. Der Pseudo-NMOS hat also einen DC-Querstrom-Anteil und keine perfekte logische 0,  $V_{0l} \neq 0V$ . Wird ein Pseudo-NMOS-Gatter nun mit einem Clock belegt, der in Precharge-Phase und Evaluate-Phase aufgeteilt ist, wird aus diesem Pseudo-NMOS Gatter ein SRDL-Gatter, siehe Abbildung 2.41. Mit dem dynamischen Ansatz können wir Schaltungen entwerfen, die schneller als statische Gatter sind und weniger Strom verbrauchen als Pseudo-NMOS-Gatter.

### 2.8.2 Funktionsweise eines DL-Gatters

Das Allgemeine SRDL aus Abbildung 2.41 wird nun in seiner Funktionsweise beschrieben. Wie bereits erwähnt wird der Clock-Zyklus in zwei Phasen aufgeteilt, einmal die Precharge-Phase und die Evaluate-Phase. In der Precharge-Phase mit 0V am Eingang

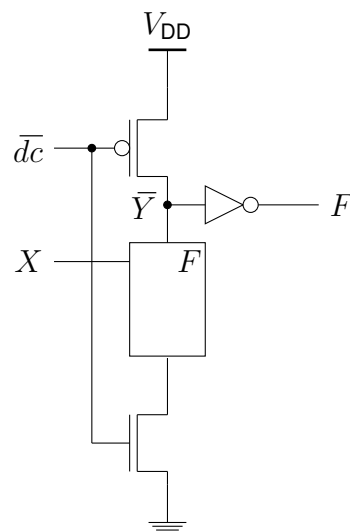


Abbildung 2.41: Allgemeines SRDL-Gatter

$\bar{d}c$  wird, wie der Name bereits vermuten lässt, der interne Knoten  $\bar{Y}$  auf  $V_{DD}$  aufgeladen, sodass eine 0 am Ausgang anliegt. Nach der Precharge-Phase wird die Evaluate-Phase eingeleitet, indem  $\bar{d}c$  auf 1 geschaltet wird, der Evaluate-NMOS wird leitend. Ist die Funktion  $F$  erfüllt, liegt eine 0 am Knoten  $\bar{X}$  an bzw. eine 1 am Ausgang  $F$ , ist die Funktion nicht erfüllt liegt eine 1 am internen Knoten  $\bar{X}$  bzw. eine 0 am Ausgang an.

### 2.8.3 Kaskadierung von DL-Gattern

Damit beim Schalten ein Domino-Effekt entsteht, müssen die DL-Gatter hintereinander geschaltet werden, dann schaltet zunächst die erste Stufe, es fällt also der erste Dominostein, und dann die zweite, bis das Ende des Pfades erreicht ist, siehe Abbildung 2.42. Im Beispiel wurde ein AND2 in der ersten und zweiten Stufe durch zwei SRDL-Gatter mit zusätzlichen Keepern realisiert. Die Keeper sind dafür da, dass am Knoten  $\bar{Y}$ , falls  $F$  nicht erfüllt ist, kein high-Z anliegt, sondern dieser Knoten in diesem Fall auch auf einem festen Potential liegt, nämlich  $V_{DD}$ . Die zugehörige Formeln in AA sind in Gleichung 2.58 und Gleichung 2.59 zu sehen zusätzlich ist das Impulsdiagramm für das Durchschalten einer 1 durch die Kaskade in Abbildung 2.43 gegeben.

$$F_0 := A \wedge B \quad (2.58)$$

$$F_1 := F_0 \wedge C \quad (2.59)$$

Es wird zunächst davon ausgegangen, dass alle Ein- und Ausgänge eine 0 anliegen hatten und in der Precharge-Phase die Eingänge  $a, b, c$  von 0 nach 1 schalten. Das Um-

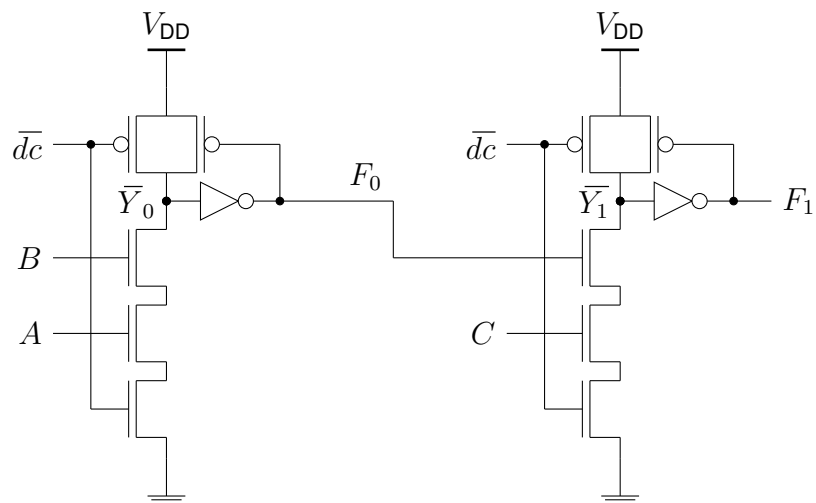


Abbildung 2.42: DL-Kaskade

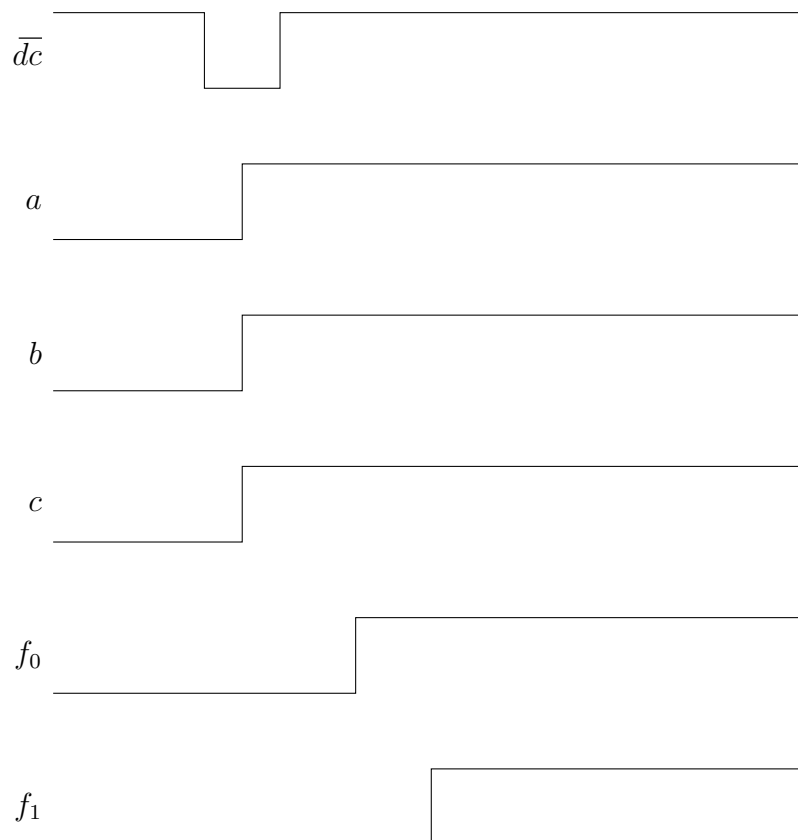


Abbildung 2.43: Impulsdiagramm der Kaskade

schalten der Eingänge wurde hier vereinfacht gleichzeitig angenommen. Dann wird an  $\bar{dc}$  eine 1 geschaltet, sodass sich zunächst der vorgeladene Knoten  $\bar{Y}_0$  entlädt, was zu einer 1 an  $F_0$  führt, was wiederum den zweiten Knoten  $\bar{Y}_1$  entlädt, wodurch schlussend-

lich eine 1 an  $F_2$  anliegt. Die Gatter werden, einschließlich des Ausgangsinverters, als Dominogatter betrachtet, da es zu Komplikationen kommt, wenn ein dynamisches Gatter ein anderes direkt speist. Da alle Ausgangsknoten während der Precharge-Phase aufgeladen werden, befinden sich die Eingänge der nachfolgenden Gatter initial auf 1. Dadurch entsteht ein aktiver Pfad zu GND, sobald der Evaluations-Transistor leitend wird. Ein einmal entladener Ausgangsknoten kann erst in der nächsten Precharge-Phase wieder aufgeladen werden, weshalb eine direkte Verbindung problematisch ist.

Der Inverter am Ausgang spielt hierbei eine zentrale Rolle. Neben dem Vorteil, dass durch den PD eine 1 erzeugt wird – was den Entwurfsprozess intuitiver macht – verhindert er, dass sich die Gatter in der Precharge-Phase gegenseitig beeinflussen. Ohne diesen Inverter könnte es passieren, dass ein nachfolgendes Gatter bereits in der Precharge-Phase aktiviert wird, was zu Fehlfunktionen führen kann. Der Inverter stellt sicher, dass die nachfolgende Stufe in der Precharge-Phase gesperrt bleibt.

Allerdings löst der Inverter nicht alle Probleme: Eine SRDL-Schaltung darf keine negativen Abhängigkeiten aufweisen, da es sonst zu einer doppelten Invertierung kommt, wodurch das ursprüngliche Problem erneut auftritt. So könnten z.B. die Gleichung 2.60 und Gleichung 2.61 nicht entworfen werden.

$$F_0 := A \wedge B \quad (2.60)$$

$$F_1 := \bar{F}_0 \wedge C \quad (2.61)$$

Um allgemeine Schaltungen entwerfen zu können, also auch mit negativen Abhängigkeiten, wird die DRDL verwendet.

#### 2.8.4 DRDL-Gatter

Werden zwei komplementäre SRDL-Gatter parallel komponiert und mit dem gleichen Evaluate-NMOS verbunden, entsteht ein DRDL-Gatter, siehe Abbildung 2.44. Hier sind zwei komplementäre SRDL-Schaltungen mit ihren entsprechenden Funktionen  $F$  und  $\bar{F}$  konstruiert. Diese Schaltungen sind parallel aufgebaut und werden vom selben Duty-Cycle  $\bar{dc}$  gesteuert. Zusätzlich haben beide noch einen Keeper. Die Verwendung von DRDL-Gatter erlaubt nun auch negative Abhängigkeiten. Ein Beispiel für die Realisierung von Gleichung 2.60 und Gleichung 2.61 ist in Abbildung 2.45 gegeben.



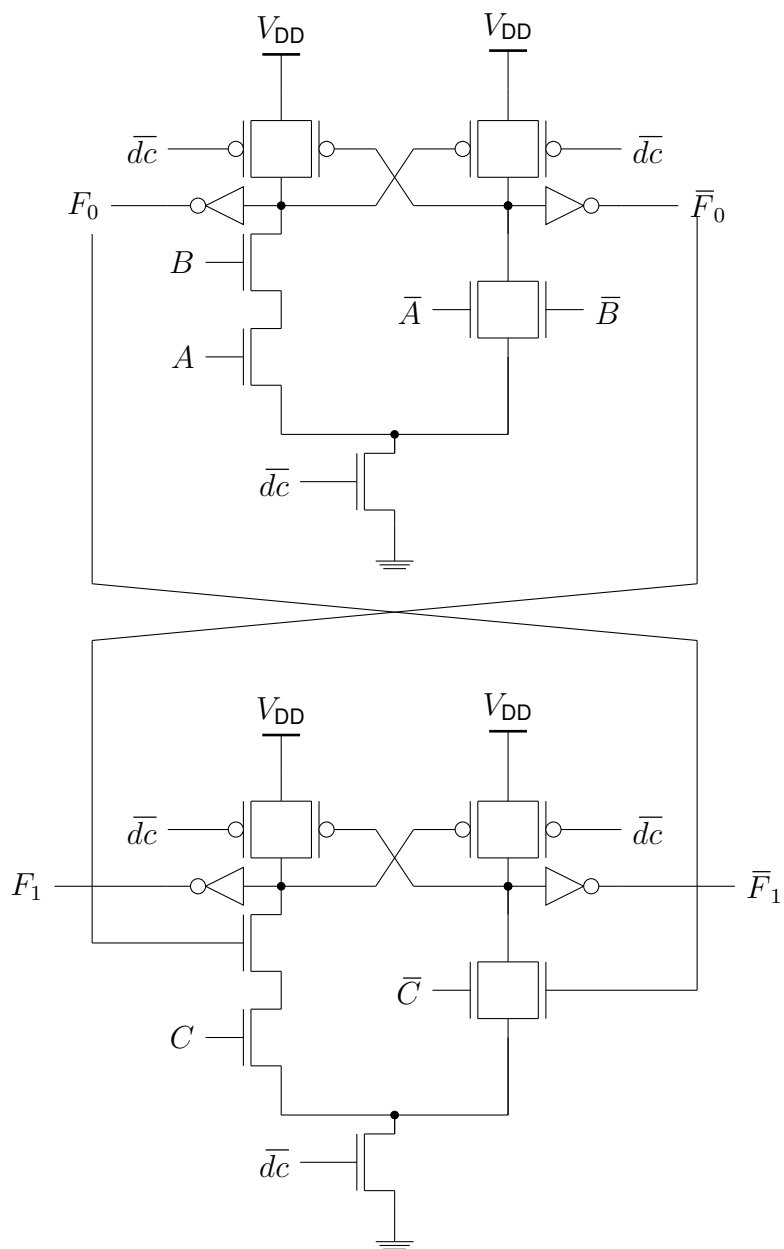


Abbildung 2.45: Kaskade aus DRDL-Gattern

## 3 Asynchroner Entwurf in handelsüblichen FPGAs

Dieses Kapitel behandelt den asynchronen Entwurf auf FPGAs. Zunächst wird eine Einführung in das Thema gegeben und die Motivation für die Nutzung von FPGAs erläutert, wobei insbesondere ihre Eignung für diese Arbeit hervorgehoben wird. Anschließend erfolgt ein Vergleich verschiedener FPGA-Hersteller sowie ihrer Frameworks zur Umsetzung asynchroner Schaltungen. Abschließend werden verschiedene asynchrone Entwurstile vorgestellt, deren Implementierung auf FPGAs analysiert und bewertet, um herauszuarbeiten, welcher Ansatz sich am besten eignet.

### 3.1 Einführung

Die VLSI-Technologie hat eine kostengünstige Möglichkeit zur Implementierung leistungsstarker digitaler Schaltungen eröffnet. Dadurch ist es möglich geworden, Chips mit mehreren Milliarden Transistoren zu entwickeln, wie es moderne Mikroprozessoren eindrucksvoll zeigen. Solche Chips werden im sogenannten Full-Custom-Ansatz gefertigt, bei dem alle Bestandteile einer VLSI-Schaltung sorgfältig angepasst werden, um die jeweiligen spezifischen Anforderungen zu erfüllen. In der Unterhaltungselektronikbranche ist es entscheidend, neue Produkte in kürzester Zeit auf den Markt zu bringen. Eine verkürzte Entwicklungs- und Produktionszeit ist daher unerlässlich. Ebenso wichtig ist es, das finanzielle Risiko bei der Entwicklung neuer Produkte zu minimieren, um mehr innovative Ideen als Prototypen realisieren zu können.

FPGAs haben sich als optimale Lösung für diese Herausforderungen hinsichtlich verkürzter Markteinführungszeit und Risikomanagement erwiesen. Sie ermöglichen es Schaltungsentwicklern, die Logikstruktur direkt zu konfigurieren, ohne auf die Fertigungsanlagen für integrierte Schaltungen angewiesen zu sein [52].

Im Vergleich zu ASICs bietet das FPGA-Design mehrere Vorteile [53]:

- **Kostengünstige Validierung und effiziente Verifikation:** FPGAs ermöglichen eine frühzeitige und iterative Verifikation des Designs direkt in der Hardware, ohne teure Fertigungsprozesse oder Maskenkosten.

- Vorteile über den gesamten Lebenszyklus: Nachträgliche Anpassungen erhöhen die Flexibilität und verlängern die Nutzungsdauer.
- Schnelle Prototypenerstellung: Neue Konzepte können direkt umgesetzt und getestet werden, was die Entwicklungszeit erheblich reduziert.
- Niedrige Kosten der EDA-Werkzeuge: Die benötigten Design- und Simulationswerkzeuge sind im Vergleich zu ASIC-Tools kostengünstiger.

Angesichts ihrer zahlreichen Vorteile werden FPGAs zunehmend als Prototyping-Tool eingesetzt, um Designs zu simulieren, zu testen und zu verifizieren. Dadurch lassen sich Fehler frühzeitig erkennen und ein zuverlässiges Endprodukt sicherstellen. Mit der fortschreitenden Entwicklung der System on Chip (SoC)-Technologie steigt ihre Bedeutung weiter. Besonders in eingebetteten Systemen ermöglicht die Rekonfigurierbarkeit von FPGAs eine flexible und effiziente Implementierung, wodurch sowohl schnelles Prototyping als auch die Anpassung an spezifische Anforderungen erleichtert werden. Darüber hinaus dienen FPGAs inzwischen nicht mehr ausschließlich als Prototyping-Lösung, sondern zunehmend auch als eigenständige Hardwarebausteine. Sie stellen eine kostengünstige Alternative für Entwurf und Realisierung dar und erlauben durch ihre Rekonfigurierbarkeit nachträgliche Anpassungen sowie Optimierungen. Damit eröffnen FPGAs vielversprechende Perspektiven für das moderne SoC-Design. Ziel der Arbeit ist es, asynchrone Schaltungen zu entwerfen, die sicher und zuverlässig sind. Als Hardware-Plattform wurden FPGAs ausgewählt, um die genannten Vorteile zu nutzen. Es werden in einer Hardware Description Language (HDL) reale Strukturen aufgebaut und als eine Art Library-Konzept von bekannten Strukturen verwendet. Im Folgenden werden die Schaltungsstruktur eines FPGA sowie die Prozesse im FPGA-Flow im Detail betrachtet.

## 3.2 Field Programmable Gate Array

Die heute weit verbreitete FPGA-Architektur basiert auf dem 1985 von Xilinx entwickelten SRAM-basierten FPGA. Eine schematische Darstellung ist in Abbildung 3.1 zu sehen. Wie dargestellt, besteht die Struktur aus einem zweidimensionalen Array von Logikblöcken (Configurable Logic Block (CLB)), die über universelle Verbindungsressourcen miteinander verbunden werden können.

Die Zwischenverbindung (engl. Interconnect) umfasst Drahtsegmente unterschiedlicher Länge. In dieser Verbindungsebene sind programmierbare Schalter integriert, die

es ermöglichen, Logikblöcke mit Drahtsegmenten oder Drahtsegmente untereinander zu verbinden. Diese werden im Folgenden Programmable Interconnect Points (PIP) genannt. Logikschaltungen werden im FPGA implementiert, indem die gewünschte Logik auf einzelne Logikblöcke aufgeteilt wird. Anschließend werden diese Blöcke mithilfe der programmierbaren Schalter nach Bedarf miteinander verknüpft [52], [54]. Eine beispielhafte Zeichnung ist in Abbildung 3.1 zu sehen.

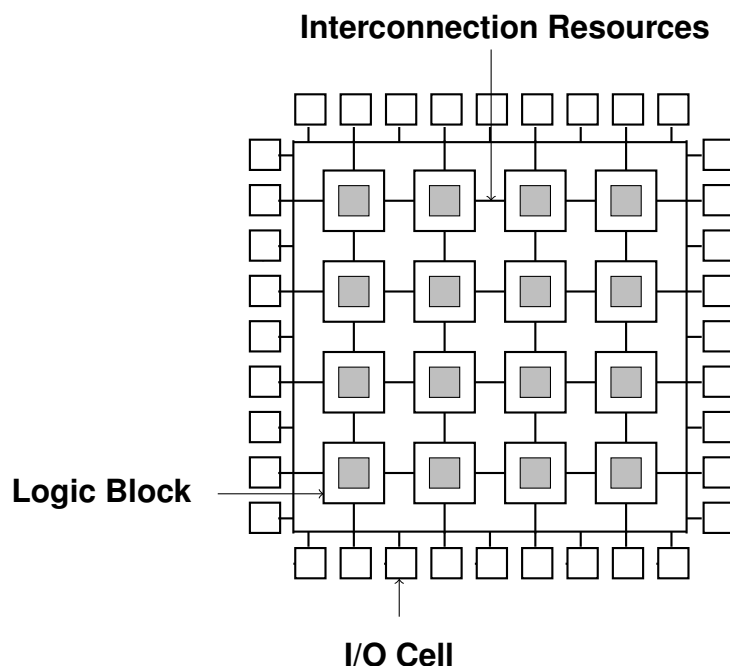


Abbildung 3.1: FPGA-Struktur

Die innere Schaltungsstruktur innerhalb eines einfachen Logikblocks [52] ist in Abbildung 3.2 dargestellt.

Ein Logikblock bildet die grundlegende Einheit zur Implementierung digitaler Schaltungen und besteht aus mehreren zentralen Komponenten.

Im Kern des Logikblocks aus Abbildung 3.2 befindet sich eine LUT, die als konfigurierbare Logikeinheit dient. Sie erhält mehrere Eingangssignale und berechnet anhand einer gespeicherten Wahrheitstabelle das entsprechende Ausgangssignal. Die LUT kann sowohl für kombinatorische als auch für einfache sequentielle Schaltungen verwendet werden. Ergänzt wird die Struktur durch ein D-Flipflop, das für die Speicherung von Zuständen genutzt wird. Über einen Taktimpuls ( $Clk$ ) kann das Flipflop das Signal

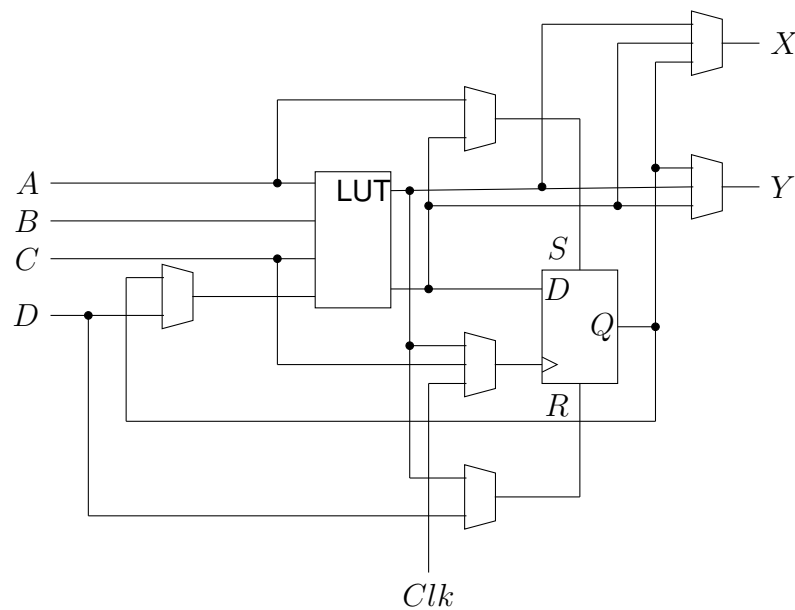


Abbildung 3.2: Struktur eines einfachen Logikblocks

zwischen speichern, sodass der FPGA sowohl synchrone als auch kombinatorische Logik realisieren kann.

Zur flexiblen Steuerung des Signalfusses sind im Logikblock mehrere user-programmierbare Multiplexer integriert. Diese ermöglichen es, gezielt zwischen verschiedenen Ein- und Ausgangssignalen zu wählen. So kann entschieden werden, ob das direkte LUT-Signal oder das gespeicherte Signal des Flipflops weiterverarbeitet wird. Durch die Programmierbarkeit der Multiplexer kann der Logikblock an spezifische Anforderungen angepasst werden, was die Vielseitigkeit des FPGA-Designs erheblich steigert. Die endgültigen Ausgangssignale des Logikblocks stehen an definierten Anschlüssen zur Verfügung und können für weitere Berechnungen oder externe Verbindungen genutzt werden.

Die Verdrahtungsstruktur zwischen den Logikblöcken ist essenziell für die Leistung der Schaltung. Besonders in asynchronen Schaltungen spielt das Verdrahtungs-Delay eine wichtige Rolle, da es die Signalintegrität und das Timing beeinflusst. Die in der Abbildung dargestellten Signalwege verdeutlichen die internen Verbindungen zwischen den Komponenten des Logikblocks.

Die Verbindungsstruktur [52], [55] im FPGA ist in Abbildung 3.3 dargestellt.

Es ist zu erkennen, dass die Verbindungen innerhalb des FPGA zwischen den CLBs über die Switch-Matrix realisiert werden. Zusätzlich existieren jedoch schnellere Direkt-

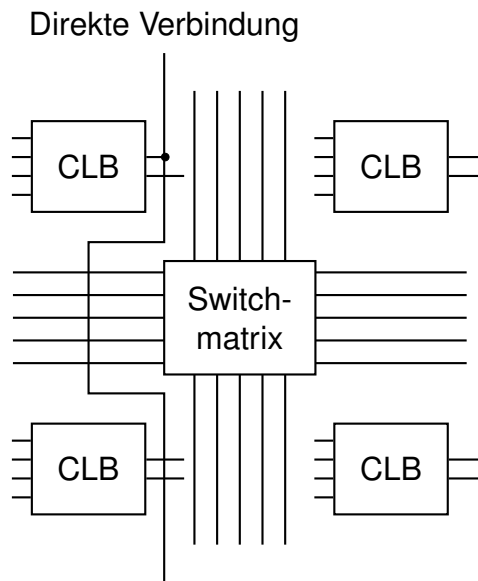


Abbildung 3.3: Zwischenverbindungen zwischen Logikblöcken im FPGA

verbindungen, die bestimmte Pfade ohne Umweg über die Switch-Matrix ermöglichen. Ein optimales Verdrahten, also das Ausbalancieren der Pfade, ist entscheidend für die Timing-Einhaltung und erfolgt während des Implementierungsschritts.

Das synthetisierte logische Design wird im FPGA durch eine Kombination aus CLBs und den dazwischen gerouteten Verdrahtungen umgesetzt. Dabei sorgt die Routing-Optimierung dafür, dass die Signale effizient verteilt werden und eine optimale Leistungsfähigkeit der Schaltung gewährleistet ist. Dies gilt natürlich primär für synchrone Schaltungen. Durch geeignete Verdrahtung und Platzierung, wie später gezeigt auch durch manuelle Platzierung, der Logikblöcke können auch asynchrone Schaltungen passend platziert werden, um Strukturazard-frei und funktionsstabil entwerfen zu können.

### 3.2.1 Look-Up Tables

Bei LUTs handelt es sich um  $n$ -stufige Multiplexer, deren Datensignale fest programmiert werden, sodass das Ausgangssignal lediglich von  $n$  Steuervariablen und somit  $2^n$  möglichen Belegungen der Steuervariablen abhängig ist. Folglich entsprechen LUTs der schaltungstechnischen Umsetzung einer Schaltbelegungstabelle üblicherweise durch Pass-Transistoren oder Transmission-Gates [56], [57]. Eine LUT2 aus Passtransistoren ist in Abbildung 3.4 dargestellt. Die Struktur umfasst die Eingang-

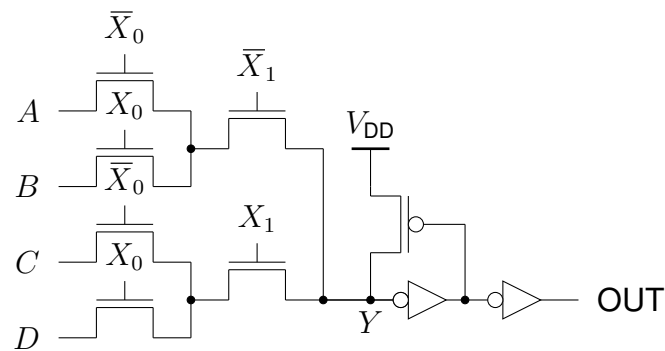


Abbildung 3.4: Zweistufiger MUX unter Verwendung von Passtransistoren

spins  $X_1$  und  $X_0$ , die so gestaltet sind, dass stets genau ein Pfad aktiv ist. Die Pins  $X_1$  und  $\bar{X}_1$  sind dabei fest mit einem Inverter verdrahtet, gleiches gilt natürlich auch für die Pins  $X_0$  und  $\bar{X}_0$ . Die feste Verdrahtung bedeutet dabei auch, dass high-Z in einer LUT niemals möglich ist, da ein Pfad auf jeden Fall durchschaltet und  $(A, B, C, D)$  mit Werten aus 1 und 0 belegt sind. Da ein NMOS-Transistor keine ideale logische 1 ( $V_{DD}$ ) treiben kann, wird am Punkt  $Y$  ein sogenannter Level Restorer eingesetzt. Dieser hebt das Signal zuverlässig auf  $V_{DD}$  an und stellt damit eine korrekte Spannungspegelung sicher. Das Propagation-Delay einer LUT bleibt dabei stets unabhängig von der zu implementierenden booleschen Funktion und ist in der Praxis vernachlässigbar gering [58].

### 3.2.2 Speicherelemente

D-Flipflop (DFF) und D-Latches sind essenzielle Speicherzellen in modernen FPGAs und ermöglichen die Umsetzung synchroner Schaltungen. Sie sind, wie bereits erwähnt, in den CLBs zu finden. Als konfigurierbares Speicherelement eignen sich LUTs [59], es sind aber auch reine taktflankengesteuerte Flipflops in FPGA-Designs zu finden. Die D-Eingänge der Flip-Flops können entweder direkt von der zugehörigen LUT oder über separate Bypass-Eingänge angesteuert werden, sodass die Flip-Flops auch als unabhängige Register genutzt werden können [60].

## 3.3 Der FPGA-Entwurfsfluss

In diesem Abschnitt wird der Entwurfsprozess von Schaltungen für FPGAs erläutert, beginnend mit der Beschreibung auf Register-Transfer-Level (RTL)-Ebene mittels

einer Hardware Description Language (HDL). Der vereinfachte FPGA-Entwurfsfluss ist schematisch in Abbildung 3.5 dargestellt [55], [61], [62].

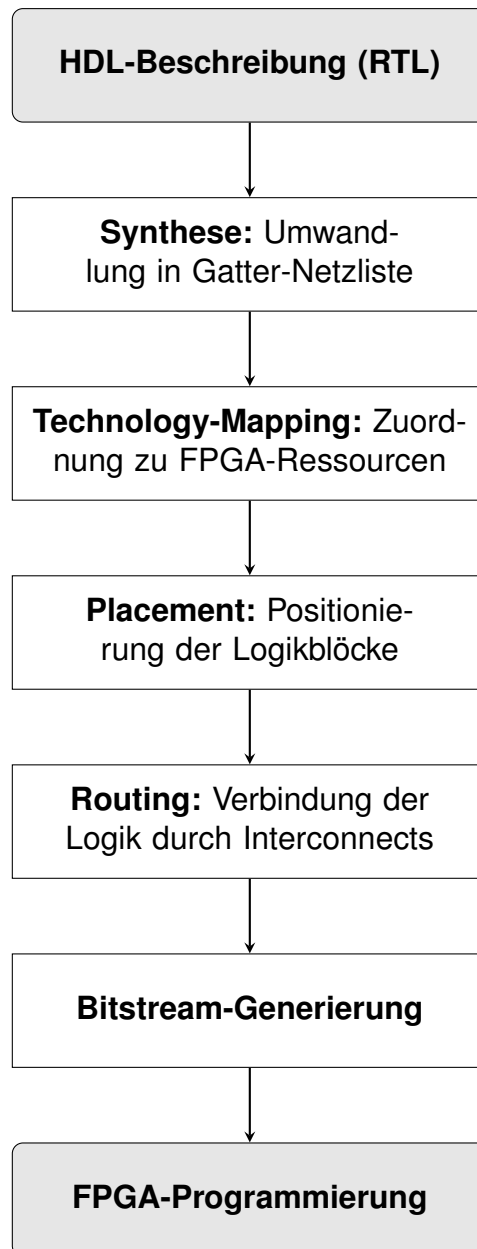


Abbildung 3.5: FPGA-Flow

Der übliche FPGA-Entwurfsfluss beginnt mit der Beschreibung der Schaltung auf RTL mithilfe einer HDL, beispielsweise Verilog HDL oder Very High-speed Integrated Circuit Hardware Description Language (VHDL). Diese Beschreibung erfolgt in der Entwicklungsumgebung des jeweiligen FPGA-Herstellers.

Im nächsten Schritt wird das RTL-Design durch einen Synthesizer in eine logische Netzliste umgewandelt, die die funktionale Implementierung in Form von logischen Gattern beschreibt. Anschließend wird im Technology-Mapping-Schritt diese Netzliste den verfügbaren FPGA-Ressourcen zugeordnet, insbesondere den LUTs und anderen konfigurierbaren Logikelementen.

Nach dem Mapping folgt das Placement & Routing:

- Beim Placement wird festgelegt, an welchen physischen Stellen im FPGA die synthetisierten logischen Elemente platziert werden.
- Das Routing bestimmt die Verbindungen zwischen diesen Elementen unter Verwendung der konfigurierbaren Interconnects des FPGA.

Im letzten Schritt wird auf Basis des platzierten und gerouteten Designs die Bitstream-Datei generiert. Diese Datei enthält die Konfigurationsdaten, die benötigt werden, um den FPGA zu programmieren. Der Bitstream definiert die Programmierung der LUTs, die Schaltung der Interconnects sowie die Aktivierung der Flip-Flops und Speicherblöcke innerhalb des FPGA.

Nach der Übertragung des Bitstreams auf das FPGA ist die gewünschte Schaltung im programmierbaren Logikbaustein implementiert und kann ausgeführt werden.

### 3.3.1 Hardware Description Languages

HDLs sind spezielle Programmiersprachen, die zur Beschreibung, Modellierung und Simulation von digitalen Schaltungen verwendet werden. Im Gegensatz zu herkömmlichen Programmiersprachen wie C oder Python sind HDLs darauf ausgelegt, die inhärent parallele Natur von Hardware zu erfassen und ihre Funktionalität auf verschiedenen Abstraktionsebenen darzustellen.

HDLs ermöglichen es Ingenieuren, komplexe digitale Designs effizient zu entwickeln, zu testen und für die Implementierung auf physischer Hardware, wie beispielsweise ASICs (Application-Specific Integrated Circuits) oder FPGAs (Field-Programmable Gate Arrays), vorzubereiten. Zu den gängigsten Hardware-Beschreibungssprachen zählen *Verilog HDL* und *VHDL*. Beide haben sich als Industriestandards etabliert und bieten umfangreiche Möglichkeiten für die Modellierung und Verifikation digitaler Systeme. [63], [64].

### 3.3.2 Vergleich von VHDL und Verilog als Hardware Description Languages

Heutzutage dominieren zwei Hardware Description Languages (HDLs) den Markt: VHDL und Verilog [63]. VHDL steht für Very High-Speed Integrated Circuit Hardware Description Language und wurde ursprünglich das US-Verteidigungsministerium entwickelt [63]. Im Jahr 1983 wurde die Hardwarebeschreibungssprache Verilog von Automated Integrated Design Systems als Sprache zur logischen Simulation entwickelt. Bemerkenswert ist, dass die Entwicklung von Verilog vollständig unabhängig vom VHDL-Projekt erfolgte [63]. Verilog ist eine kompaktere Sprache, die sich stärker an der Syntax von Programmiersprachen wie C orientiert. Beide HDLs sind in der digitalen Schaltungsentwicklung weit verbreitet und werden sowohl für FPGA- als auch Anwendungsspezifische integrierte Schaltung (ASIC)-Designs genutzt.

Ein wesentlicher Unterschied zwischen den beiden Sprachen liegt in ihrer Syntax und Typisierung. Während Verilog eine eher flexible Handhabung von Datentypen erlaubt, setzt VHDL auf eine strikte Typisierung und explizite Signaldeklaration [63]. Diese Eigenschaft macht VHDL weniger fehleranfällig und erleichtert die Lesbarkeit und Wartbarkeit von Schaltungen, insbesondere in großen und komplexen Designs [65].

VHDL zeichnet sich zudem durch einen größeren Funktionsumfang aus [64].

Diese Vorteile führen dazu, dass VHDL insbesondere in sicherheitskritischen Anwendungen bevorzugt wird, etwa in der Luft- und Raumfahrt, der Automobilindustrie oder der Medizintechnik [66].

Ein weiteres Argument für die Nutzung von VHDL ist sein didaktischer Wert. Da die strikte Syntax Fehler frühzeitig erkennt und die Strukturierung von Schaltungen erzwingt, eignet sich VHDL besonders für den Einstieg in die digitale Schaltungsentwicklung [63]. Dies macht es zu einer sinnvollen Wahl für die Einführung in Hardwarebeschreibungssprachen in der akademischen Lehre sowie in industriellen Ausbildungsprogrammen.

### 3.3.3 Logiksynthese

Nachdem die gewünschte Funktion des Systems mittels HDL-Code beschrieben wurde, erfolgt die Übergabe an den Logiksynthesizer. Innerhalb des Syntheseprozesses durchläuft das Design mehrere iterative Optimierungsschritte, bevor schließlich die Netlist-Dateien der implementierten Schaltung generiert werden. Während der Synthese werden verschiedene Prüfungen und Optimierungen durchgeführt, um eine

effiziente und fehlerfreie Implementierung sicherzustellen [67]:

- **Hierarchieerhaltung:** Strukturierung und Modularität des Designs bleiben erhalten.
- **Einfügen von I/O-Buffern:** Anpassung an physikalische Pins zur Signalintegrität.
- **Kodierung der endlichen Zustandsautomaten (FSMs):** Optimierung von Zustandsübergängen und Reduzierung des Logikbedarfs.
- **Umformung und Nutzung eingebetteter Komponenten:** Implementierung optimierter FPGA-spezifischer Bausteine (z. B. DSP-Blöcke, Speicher).
- **Beschränkung des maximalen Fan-Outs:** Reduktion der Last pro Signal zur Einhaltung von Timing-Anforderungen.
- **Register-Duplizierung, Retiming und Balancierung:** Verbesserung der Taktverteilung und Minimierung von Verzögerungen.

### 3.3.4 Implementierungsprozess

Nachdem der HDL-Code erfolgreich in eine Netlist konvertiert wurde, erfolgt im Implementierungsprozess eine weitere Optimierung, die speziell an die Architektur des FPGAs angepasst ist. Dieser Prozess besteht aus zwei zentralen Schritten: Technology-Mapping sowie Platzierung und Verdrahtung (Placement & Routing).

Zunächst wird die aus dem Synthesizer generierte Netlist, die bereits grundlegende Optimierungen enthält, im Technology-Mapping weiterverarbeitet. Dabei wird die optimierte logische Schaltungsstruktur in das Netz der konfigurierbaren Logikblöcke (CLBs) des FPGAs abgebildet. Anschließend werden die CLB-Schaltungen in die physikalische FPGA-Hardware überführt:

- **Platzierung (Placement):** Die Positionen der Logikblöcke, die den logischen Funktionen entsprechen, werden innerhalb des FPGA-Gitters bestimmt.
- **Verdrahtung (Routing):** Die Verbindungen zwischen den Logikblöcken werden durch programmierbare Interconnects und die Switch-Matrix realisiert.

Die Timing-Constraints spielen in diesem Prozess eine entscheidende Rolle: Die beiden Schritte werden iterativ durchlaufen, um eine optimierte Implementierung mit minimalen Verzögerungen und maximaler Timing-Effizienz zu erreichen.

### 3.3.5 Bitstream-Generierung

Nach Abschluss des Implementierungsprozesses wird die Schaltungsstruktur des Designs im FPGA durch Netlist-Dateien sowie Konfigurationsinformationen beschrieben. Diese Daten dienen als Grundlage für die Generierung der Bitstream-Datei, welche die endgültige Konfigurationsinformation für das FPGA enthält.

Nachdem der FPGA mit dem generierten Bitstream programmiert wurde, kann die realisierte Schaltung getestet werden. Dieser Schritt umfasst zwei zentrale Aspekte:

- **Funktionale Validierung:** Überprüfung, ob die implementierte Schaltung die spezifizierten Funktionen korrekt ausführt.
- **Schaltungsverifikation:** Kontrolle der Implementierung hinsichtlich ihrer Timing-Anforderungen und Signalintegrität.

Durch diesen abschließenden Testprozess wird sichergestellt, dass die Hardware-Implementierung den konzipierten Entwurfsspezifikationen entspricht.

## 3.4 Gegenüberstellung der Frameworks

### 3.4.1 Hintergrund

Mit der fortschreitenden Entwicklung der Halbleitertechnologie steigt die maximale Arbeitsfrequenz von FPGA-Systemen signifikant. Dadurch gewinnt das Propagation-Delay kombinatorischer Schaltungen zunehmend an Bedeutung [68]. Tritt ein Hazard auf, können die Timing-Anforderungen nicht mehr eingehalten werden, was zu schwerwiegenden Systemfehlern führt – insbesondere in FPGA-Architekturen, deren logische Bauteile auf LUTs basieren.

Zusätzlich steigt mit der zunehmenden Nutzung von FPGA-basierten SoC-Systemen auch das Risiko von Fault Injection Attacks [69], wie z. B. Glitch-Analysis. Solche Angriffe können die Funktionalität des Systems beeinträchtigen und zu Betriebsunterbre-

chungen oder Speicherüberläufen führen. Im schlimmsten Fall können sicherheitskritische Daten offengelegt werden, was die Systemintegrität erheblich gefährdet. Eine der Hauptursachen hierfür liegt im intrinsischen Delay der LUTs.

Aufgrund der physikalischen MOS-Struktur innerhalb der LUTs variieren die Eingangs- zu-Ausgangs-Propagationsverzögerungen, da die Signalpfade von unterschiedlichen Eingangspins physikalisch unterschiedlich aufgebaut sind [56]. Diese Delay-Differenzen zwischen den Signalpfaden können unerwünschte Glitches am Ausgang während einer Transition verursachen. Wird ein solcher Glitch von einer sequentiellen Schaltung fälschlicherweise abgetastet und gespeichert, kann dies zu einer Fehlfunktion des gesamten Systems führen.

Zur Minimierung dieser Effekte können Optimierungen im Syntheseprozess des FPGA-Flows eingesetzt werden. Durch gezielte Anpassungen der Technology-Mapping-Strategie kann die ursprüngliche Implementierung so modifiziert werden, dass der Einfluss des intrinsischen LUT-Delays ausgeglichen wird und Funktions hazards reduziert werden.

### **Zielsetzung dieser Arbeit**

Diese Arbeit adressiert den Synthese- und Technology-Mapping-Prozess in FPGAs mit dem Ziel, eine zuverlässige Implementierung sicherzustellen. Dafür werden zunächst verschiedene FPGA-Hersteller und ihre jeweiligen Entwurfstools analysiert und hinsichtlich ihrer Möglichkeiten zur asynchronen Schaltungsentwicklung verglichen. Im Folgenden werden dann sichere Entwurfsmethoden für asynchrone Schaltungen vorgestellt.

#### **3.4.2 Intel Altera**

In diesem Abschnitt werden die FPGA-Architektur Cyclone II von Intel Altera sowie das zugehörige Entwurfstool Quartus II vorgestellt. Die innere Struktur des Cyclone II [70] ist in Abbildung 3.6 dargestellt.

Die Abbildung zeigt die hierarchische Struktur des Cyclone II FPGA. Die Logic Array Block (LAB)s sind in einer Reihen- und Spaltenanordnung organisiert, um eine effiziente Verbindung zwischen den einzelnen Blöcken zu ermöglichen. Innerhalb eines LABs existieren Local Interconnects, die eine schnelle Kommunikation zwischen den enthal-

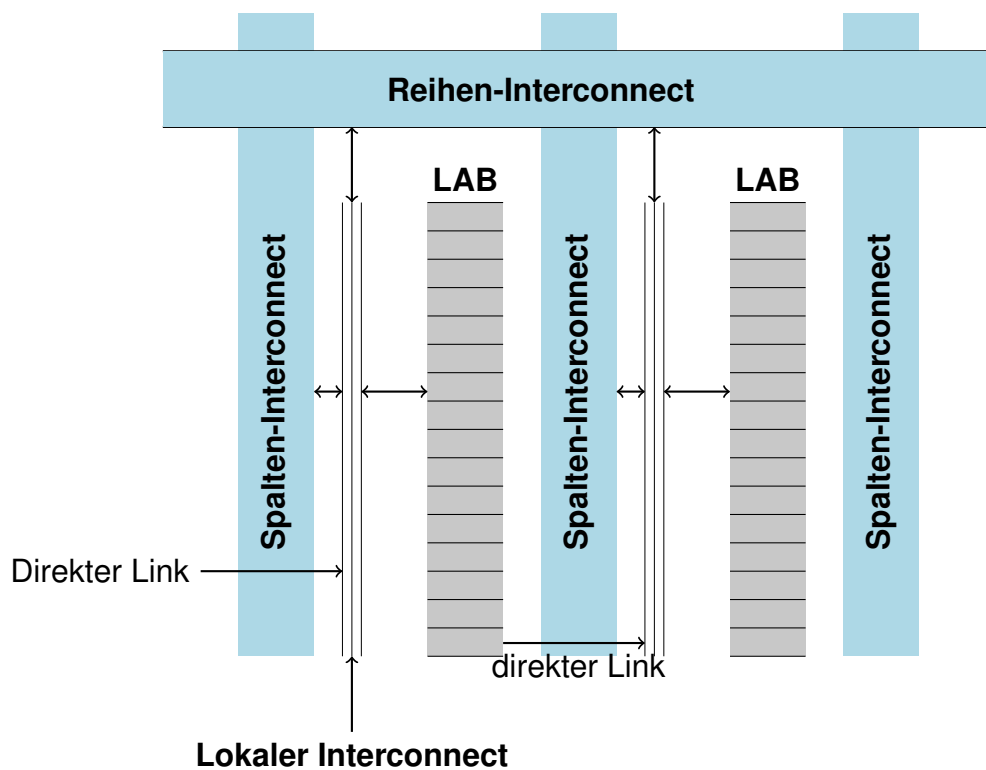


Abbildung 3.6: Interconnect-Struktur des Cyclone II FPGA

tenen Logic Element (LE)s ermöglichen. Für die direkte Verbindung zwischen benachbarten LABs werden Direct Link Interconnects verwendet, die eine schnelle Datenübertragung ermöglichen, ohne das globale Routing zu beanspruchen. Die Row- und Column-Interconnects bilden schließlich das globale Routing-Netzwerk, das Signale über größere Entfernungen innerhalb des FPGA verteilt.

### 3.4.2.1 Logikelemente und Logik-Array-Blocks

Die kleinste logische Einheit in der Cyclone II-Architektur, das LE, ist kompakt aufgebaut und bietet erweiterte Funktionen bei effizienter Nutzung der verfügbaren logischen Ressourcen. Jedes LE enthält eine vierstufige LUT, die als Funktionsgenerator dient und jede beliebige Funktion mit bis zu vier Eingangsvariablen realisieren kann. Zusätzlich verfügt es über ein programmierbares Register, das zur Speicherung von Zuständen oder Zwischenergebnissen genutzt werden kann.

Zur Unterstützung arithmetischer Berechnungen besitzt das LE eine Carry-Chain-Verbindung, die eine effiziente Implementierung von Additionen und anderen mathematischen Operationen ermöglicht. Darüber hinaus ist eine Register-Chain-Verbindung

vorhanden, die das serielle Weiterreichen von Registerwerten zwischen benachbarten LEs erleichtert. Das LE ist in der Lage, Signale an verschiedene Interconnect-Typen weiterzuleiten, darunter lokale Interconnects sowie Row-, Column-, Register-Chain- und Direct-Link-Interconnects. Dies ermöglicht eine flexible Signalführung innerhalb des FPGAs und erleichtert die Anbindung an andere logische Einheiten.

Zusätzlich unterstützt das LE Register Packing, wodurch mehrere Register innerhalb eines einzelnen LEs effizient genutzt werden können, sowie Register Feedback, das Rückkopplungen innerhalb der Schaltung erlaubt. Durch diese vielseitigen Funktionen stellt das Logic Element eine leistungsfähige und flexible Grundkomponente in der Cyclone II FPGA-Architektur dar.

Ein LAB besteht aus 16 LEs. Innerhalb eines LAB sind die LEs und deren zugehörige Register miteinander intern verknüpft, um eine effiziente logische Verarbeitung zu ermöglichen.

#### **3.4.2.2 Altera Quartus II und FPGA-Implementierung**

Altera Quartus II ist eine von Altera entwickelte Software zur Entwicklung und Implementierung digitaler Schaltungen auf FPGAs. Nach der Übernahme von Altera durch Intel wurde Quartus II unter der Bezeichnung Intel Quartus Prime weitergeführt. Die Entwicklungsumgebung ermöglicht die Analyse und Synthese von HDL-Designs, die Durchführung von Timing-Analysen, die Simulation von digitalen Schaltungen sowie die Konfiguration von FPGA-Bausteinen. Dabei unterstützt Quartus II sowohl VHDL als auch Verilog als Hardwarebeschreibungssprachen.

#### **3.4.2.3 Synthese und Mapping in Quartus II**

Der Syntheseprozess in Quartus II besteht aus mehreren aufeinanderfolgenden Schritten. Zunächst wird das durch VHDL oder Verilog beschriebene Design in eine Netlist überführt, die eine logische Beschreibung der Schaltung enthält. Im anschließenden Technology-Mapping werden die in der Netlist definierten Schaltungsstrukturen in FPGA-spezifische Hardware-Ressourcen abgebildet. Dies umfasst insbesondere die Zuordnung von logischen Funktionen zu den Logic Elements (LEs) innerhalb der Logic Array Blocks (LABs) des FPGA.

Zur Optimierung der Implementierung führt Quartus II verschiedene Anpassungen durch, darunter:

- **Logische Minimierung:** Reduktion der Gatteranzahl durch Boolesche Optimierung.
- **Register-Packing:** Effiziente Platzierung von Registern innerhalb der LEs zur optimalen Ressourcennutzung.
- **Optimierung der Carry-Chain:** Effiziente Umsetzung arithmetischer Operationen zur Reduktion der Verzögerung.
- **Platzierung und Routing:** Bestimmung der optimalen Positionierung der LEs und der Verbindungen zwischen ihnen.

#### 3.4.2.4 Visualisierung der FPGA-Architektur

Quartus II bietet verschiedene Analysetools zur Visualisierung der implementierten Schaltungsstruktur. Der Technology Map Viewer stellt das Schaltbild auf LUT-Ebene dar und ermöglicht eine detaillierte Analyse der Synthese-Ergebnisse. Dabei können die erzeugten LUT-Strukturen überprüft und deren Wahrheitstabellen eingesehen werden.

Für weiterführende Analysen auf Hardware-Ebene ermöglicht der Chip Planner eine grafische Darstellung der physischen Platzierung der LEs innerhalb des FPGA. Dies ist insbesondere bei zeitkritischen Designs und der Optimierung von Signalpfaden relevant. Durch eine gezielte Platzierung der LEs innerhalb eines LABs können Hazards minimiert und Timing-Constraints optimiert werden.

#### 3.4.2.5 Manuelle Platzierung der LUTs und Festlegung der Eingangspins

Bei asynchronen Schaltungsdesigns ist eine präzise Platzierung der LUTs entscheidend, um Signalverzögerungen und Hazards zu minimieren. Eine unsachgemäße Verteilung der LUTs über mehrere LABs kann zu Timing-Problemen führen, da die Signalpfade unterschiedliche Verzögerungen aufweisen. Zur Optimierung der Signalwege sollten alle für eine bestimmte Funktion benötigten LUTs innerhalb desselben LABs positioniert werden.

Hierfür werden Schaltungen nicht durch die abstrakte Version einer HDL realisiert, sondern die einzelnen LUTs instanziiert und mit den Ausgangswerten der Tabelle vorprogrammiert. Eine einfache Instanzierung eines Muller-C ist in Listing 3.1 zu finden.

```
module MullerC ( A , B , C ) ;
input A , B;
output wire C;
//
cycloneii_lcell_comb w
(
. dataa ( A ),
. datab ( B ),
. datac ( C ),
. datad ( 1'b1),
. combout ( C )
);
defparam w . shared_arith = "off";
defparam w . extended_lut = "off";
defparam w . lut_mask =16'HE8E8;
defparam w . dont_touch = "on";
endmodule
```

*Listing 3.1: Low-Level LUT-Implementierung des Muller-C-Elements*

Quartus II ermöglicht nun eine gezielte manuelle Platzierung von LEs und Registern durch die Nutzung des Quartus Settings File (QSF). Über spezifische Platzierungsbefehle können LUTs und Register festen Koordinaten auf dem FPGA zugewiesen werden:

- `set_location_assignment LAB_X6_Y6 -to w`  
→ Weist die LUT mit der Bezeichnung `w` dem LAB an der Position `X6, Y6` zu.
- `set_location_assignment LCCOMB_X6_Y6_N0 -to w`  
→ Platziert die LUT in einer spezifischen logischen Zelle innerhalb des LABs.
- `set_location_assignment LCFF_X1_Y6_N7 -to x~reg0`  
→ Weist das Register mit der Bezeichnung `x~reg0` einer bestimmten Position zu.

Da die Signalverzögerung stark von der physischen Platzierung der LEs abhängt, muss insbesondere bei asynchronen Designs sichergestellt werden, dass kritische Signale möglichst kurze Signalpfade haben und keine unnötigen Routing-Verzögerungen auftreten.

Zudem ist auch die Eingangspinbelegung der LUTs wichtig, um die Latenz der Signale richtig zuzuordnen. Es ist besonders wichtig, dass auf die Zeitverzögerung der Rückkopplung geachtet wird.

Es wurden Versuche angestellt, die Pins zu fixieren, jedoch wurden diese im Fitting umverteilt. Es gibt auch hierfür aufwendige Lösungsansätze, indem das Routing-File manuell geändert wird, diese erfordern aber ein höchstes Maß an manuellem Routing und sind damit für größere Entwürfe nicht zu gebrauchen. Da die Pinbelegung aber essentiell ist und auch ein einfacher Entwurfsansatz angestrebt wurde, hat sich Quartus II als weniger gute Lösung für full-custom Designs herausgestellt.

### 3.4.3 Xilinx

Da die Anpassungsmöglichkeiten in Quartus II begrenzt waren und mit erheblichem Aufwand verbunden sind, wurde der Wechsel von Intel Altera zu Xilinx vollzogen. Xilinx bietet mit dem Artix-7 einen kostengünstigen, aber leistungsstarken FPGA an. Zudem stellt die Vivado Design Suite ein leistungsfähiges Entwicklungstool mit umfangreichen Funktionen bereit, das eine effizientere und flexiblere FPGA-Entwicklung ermöglicht.

#### 3.4.3.1 Architektur des Artix-7 FPGAs

Der Artix-7 FPGA basiert auf einer hierarchischen Struktur aus CLBs, die in Spalten angeordnet und in Paare sogenannter Slices unterteilt sind. Diese Slices besitzen keine direkte Verbindung untereinander. Jeder Slice enthält vier LUTs sowie acht speichernde Elemente, die für die Realisierung synchroner und asynchroner Logik genutzt werden können. Dieser Abschnitt basiert zum Großteil auf [60].

#### 3.4.3.2 Lookup-Tables (LUTs)

Die in Artix-7 FPGAs verbauten LUTs besitzen sechs Eingänge (A1-A6) und zwei Ausgänge (O5, O6). Damit können sie beliebige boolesche Funktionen mit bis zu sechs Eingangsvariablen realisieren, wobei in diesem Fall nur der Hauptausgang  $O_6$  genutzt wird.

Zusätzlich besteht die Möglichkeit, zwei Funktionen innerhalb einer LUT zu implementieren, wenn:

- beide Funktionen dieselben fünf Variablen nutzen und durch O5 und O6 getrennt ausgegeben werden, oder
- eine Funktion maximal drei und eine zweite Funktion maximal zwei Variablen besitzt; auch hier werden die Funktionen über die beiden Ausgänge realisiert.

Diese optimierte Nutzung erlaubt es, die vorhandenen LUTs effizienter für parallele Logikoperationen zu verwenden.

### 3.4.3.3 Speicherelemente und Flip-Flops

Jeder Slice enthält acht Speicherelemente, von denen vier entweder als flankengesteuerte DFFs oder als taktgesteuerte Latches<sup>1</sup> konfiguriert werden können. Die restlichen vier Speicherelemente sind immer als DFFs ausgeführt.

Jedes Speicherelement innerhalb eines Slices wird von einem gemeinsamen Steuerungssatz beeinflusst:

- **Clock (CLK):** Der globale Takteingang kann pro Slice invertiert werden, um zwischen steigender und fallender Flanke zu wählen.
- **Clock Enable (CE):** Aktiviert oder deaktiviert die Flip-Flops innerhalb des Slices.
- **Set/Reset (SR):** Kann als synchrones oder asynchrones Signal konfiguriert werden.

Dabei ist zu beachten, dass die Synchronität bzw. Asynchronität des Set/Reset-Signals nicht individuell für jedes Speicherelement innerhalb eines Slices festgelegt werden kann.

### 3.4.3.4 Timing-Eigenschaften

Die Flip-Flops der Artix-7 FPGAs sind für hohe Taktraten optimiert. Die wichtigsten Timing-Parameter sind [71]:

- **Clock-to-Q-Verzögerung:** 0,4–0,6 ns (abhängig vom Speed Grade).
- **Setup-Zeit ( $T_{SU}$ ):** ca. 0,07–0,11 ns.
- **Hold-Zeit ( $T_H$ ):** ca. 0,12–0,18 ns.

Diese kurzen Latenzen ermöglichen hohe Betriebsfrequenzen bis in den niedrigen GHz-Bereich.

---

<sup>1</sup>active-low

### 3.4.3.5 Low-Power-Optimierungen

Die Architektur der Flip-Flops wurde hinsichtlich Energieeffizienz verbessert. Ein wesentliches Feature ist das intelligente Clock-Gating:

- Falls Clock Enable (CE) deaktiviert ist, wird der Takt direkt vor den Flip-Flops gesperrt.
- Dadurch werden dynamische Leistungsverluste reduziert, ohne explizite Taktgating-Logik implementieren zu müssen.
- Diese Technik kann typischerweise eine **Leistungsersparnis von 18–30%** erreichen [72].

Zusätzlich sorgt die Verwendung des **28nm High-Performance Low-Power (HPL)** Fertigungsprozesses für eine weitere Reduktion des Stromverbrauchs [73].

### 3.4.3.6 Manuelle Platzierung der LUTs und Festlegung der Eingangspins

Auch in Vivado können die LUTs manuell platziert werden. Zunächst wird beispielhaft eine Muller-C Realisierung auf Low-Level gezeigt siehe Listing 3.2.

```
LUT6_mullerC : LUT6
generic map (
INIT => X"FFFF_0F0F_0F0F_0000") -- Specify LUT Contents
port map (
O => C, -- LUT output (1-bit)
I0 => '1', -- LUT input (1-bit)
I1 => '1', -- LUT input (1-bit)
I2 => '1', -- LUT input (1-bit)
I3 => A, -- LUT input (1-bit)
I4 => B, -- LUT input (1-bit)
I5 => C-- LUT input (1-bit)
);
```

*Listing 3.2: Low-Level LUT-Implementierung des Muller-C-Elements*

Diese LUT kann nun auch wie in Quartus manuell platziert werden. Dies geschieht im Constraints-File mit den folgenden Befehlen:

- `set_property LOC SLICE_X0Y0 [get_cells CUT/LUT6_MullerC]`  
→ Weist die LUT mit der Bezeichnung `LUT6_MullerC` dem Slice an der Position `X0, Y0` zu.

- `set_property BEL A6LUT [get_cells CUT/LUT6_RSBuffer]`  
→ Platziert die LUT an die Position A des SLICE\_X0Y0.
- `set_property LOCK_PINS {I0:A1 I1:A2 I2:A3 I3:A4 I4:A5 I5:A6} [get_cells CUT/LUT6_RSBuffer]`  
→ Setzt die Pins I0 bis I5 der Low-Level-LUT-Instanziierung den festen physikalischen Pins A1 bis A6 der realen LUT zu.

Zwei zusätzliche Befehle haben sich als entscheidend herausgestellt: Erstens die Möglichkeit, kombinatorische Schleifen in die Constraints einzubinden, und zweitens die Don't-touch-Befehle, um zu verhindern, dass die VDS irgendwelche Einstellungen modifiziert. Die kombinatorischen Schleifen werden in den Constraints erlaubt, indem auf die logische Zelle mit der asynchronen Rückkopplung verwiesen wird, siehe Listing 3.3.

```
set_property ALLOW_COMBINATORIAL_LOOPS true [get_nets -of_objects [
  get_cells CUT/LUT6_MullerC]]
```

*Listing 3.3: Combinatorial Loops erlauben*

Die Don't-touch-Befehle werden in der Instanziierung der Komponenten geschrieben, siehe Listing 4.3.

```
component MullerC is
  port (
    A : in std_logic;
    B : in std_logic;
    C : out std_logic;
  );
end component;
attribute dont_touch of MullerC : component is "yes";
```

*Listing 3.4: Don't-touch-Befehl*

### 3.4.4 Fazit

Basierend auf der Analyse der FPGA-Entwicklungsumgebungen von Intel Altera und Xilinx wurde Xilinx als bevorzugte Wahl für die Implementierung asynchroner Schaltungen ausgewählt. Die Entscheidung basiert auf mehreren wesentlichen Faktoren:

- **Bessere manuelle Platzierung:** Xilinx Vivado erlaubt eine einfachere Kontrolle über die Platzierung von LUTs und Registern. Dies ist entscheidend für die Vermeidung von Timing-Problemen in asynchronen Designs.

- **Effektivere Constraints:** In Vivado lassen sich Platzierungs- und Routing-Constraints gezielt setzen, sodass automatische Optimierungen, die das asynchrone Verhalten beeinträchtigen könnten, vermieden werden.
- **Zuverlässige Don't-touch-Attribute:** Während Quartus II Platzierungsbeschränkungen oft ignoriert oder umverteilt hat, ermöglicht Vivado eine stabile Fixierung von den Pins der LUTs ohne ungewollte Modifikationen durch den Synthesizer.

Quartus II von Intel Altera stellte sich als weniger geeignet für Full-Custom-Designs heraus, da die manuelle Platzierung und das Routing von LUTs und deren Pins mit erheblichem Aufwand verbunden sind. Zudem wurden Pin-Fixierungen im Fitting-Prozess automatisch verändert, was zu unerwünschten Designanpassungen führte.

Xilinx hingegen bietet mit Vivado eine leistungsstarke und flexible Umgebung, die eine gezielte Platzierung und Optimierung erlaubt. Aufgrund dieser Vorteile wurde Xilinx für die Entwicklung asynchroner Schaltungen als die bessere Wahl identifiziert, und in dieser Thesis wurde der weitere Entwurfsprozess darauf aufgebaut. Die Schaltungen werden im Folgenden mit dem Digilent Arty A7 Development Board realisiert, welches einen Artix-7 (xc7a35ticsg324-1L) FPGA verwendet. Der FPGA beinhaltet 5200 Logic Slices [74].

### 3.5 Realisierung eines funktionsstabilen Muller-C-Elements im Artix-7

Im letzten Abschnitt dieses Kapitels wird die Vivado Design Suite (VDS) und deren Freiheitsgrade im manuellen Festlegen von den Eingangspins der LUTs verwendet, um ein funktionsstabiles Muller-C-Element zu realisieren. Da LUTs kein high-Z innerhalb ihrer Struktur erzeugen können, kann keine Zustandsstabilisierung im FPGA implementieren werden, daher wird das Muller-C funktionsstabil im FPGA entworfen werden.

$$\tau_{\delta} > \tau_{\Delta} \tag{3.1}$$

Wichtig ist hier die Ungleichung Gleichung 3.1 zu erfüllen [24], sodass der Vorwärtspfad des Muller-C, siehe Abbildung 3.7, eine höhere Verzögerung als der Rückkoppelpfad aufweist.

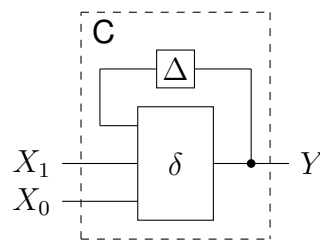


Abbildung 3.7: Muller C-element

Für die Realisierung ist das Festlegen der Eingangspins der logischen Elemente essentiell, da so gewährleistet werden kann, dass die Ungleichung der beiden Pfade erfüllt wird. Ziel ist es, jeden Eingang manuell festzusetzen, um die richtigen Delays einzustellen.

Zunächst wird die Struktur im FPGA betrachtet. Anschließend erfolgt eine Messung der Eingangspins der LUTs, um zu bestimmen, welche die schnellsten sind. Das Ziel ist es, ein Muller-C-Element, siehe Abbildung 2.16, als elementaren Baustein für das asynchrone Design funktional zu verifizieren und die Erkenntnisse aus dem Entwurfsprozess auf die weitere Entwicklung anzuwenden.

### 3.5.1 Entwurf des funktionsstabilen Muller-C Elements

Zur Implementierung des Muller-C wird eine LUT verwendet, wobei der Ausgang auf den Eingang [75] rückgekoppelt wird.

Wir betrachten nun eine Multiplexer-Struktur [76] mit sechs Eingängen zur Realisierung einer Wahrheitstabelle mit  $2^6$  Einträgen, siehe Abbildung 3.8. Diese Struktur entspricht

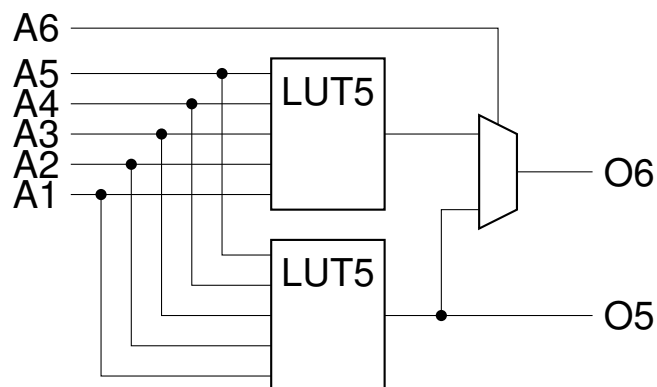


Abbildung 3.8: Struktur einer 6-Input LUT der Xilinx 7 Series FPGAs

der 6-Eingangs-LUT der Xilinx 7-Serie. Sie hat zwei verschiedene Ausgänge  $O_5$  und  $O_6$ . Aus der Strukturbetrachtung in Abbildung 3.4 ist zu erwarten, dass die Verwendung des Eingangs A6 für die Rückkopplung die niedrigste Verzögerung ergibt, da der Eingang A6 dem Ausgang am nächsten liegt.

### 3.5.2 Messung der Verzögerung des Rückkopplungspfads

Das Feedback Loop Path Delay (FLPD) ist gleich der Summe der Vorwärtswegverzögerung  $\tau_\delta$  und der Rückkopplungsverzögerung  $\tau_\Delta$  und hängt vom verwendeten LUT-Eingangspin ab. Um die Funktionsstabilisierung zu gewährleisten und die Schaltgeschwindigkeit des Müller-C-Elements zu maximieren, wird der Eingang der LUT, der dem kleinsten FLPD entspricht, durch Messung bestimmt.

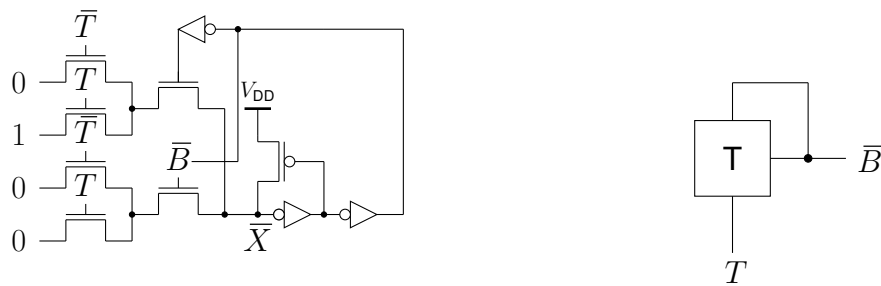
#### T-Buffer

Für die Messung wird die logische Funktion der rückgekoppelten LUT so geändert, dass der T-Buffer (TB) entsteht. Der Ausgang  $B$  der LUT wird auf den Eingang zurückgekoppelt, so dass der neue Ausgangszustand invers vom alten Ausgangszustand abhängt. Wenn die LUT gestartet (ge-triggered) wird, wird der neue invertierte Ausgangszustand auf den geschalteten Wert des alten Ausgangszustandes gesetzt. Infolgedessen hat der TB einen oszillierenden Zustand mit einer Periode, die dem Doppelten der FLPD entspricht. Die Periode oder Frequenz der Oszillation hängt davon ab, welcher Eingangspin zur Rückkopplung des Ausgangs verwendet wird. Die Wahrheitstabelle für den Ausgang  $B$  des TB findet sich in Tab. 3.1, die Formel des TB lautet  $\bar{B} = T \wedge B$ . Abbildung 3.9(a) zeigt die schematische Darstellung des TB in TL. Um eine konsisten-

*Tabelle 3.1: Phasenliste des T-Buffer*

$\bar{B}$	$T$	$B$	Comment
-	0	0	Reset
0	1	1	Oscillation
1	1	0	Oscillation

te und strukturelle Analyse für delay-insensitive Schaltungen zu ermöglichen, wurde das Symbol des TB wie in Abbildung 3.9(b) dargestellt gewählt.



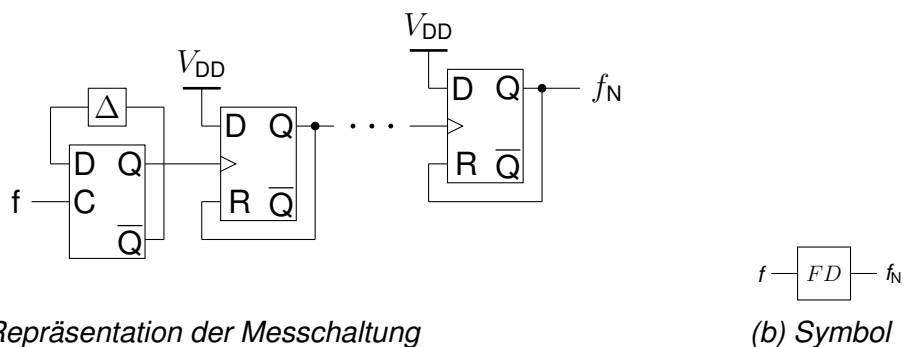
(a) Schematic (TL)

(b) Symbol (LL)

Abbildung 3.9: T-Buffer auf TL und GL

## Messschaltung

Nach den Zeitverzögerungsschätzungen von Vivado dürfte die FLPD des TB im Bereich von Hunderten von Pikosekunden liegen. Daraus ergeben sich Frequenzen von mehreren Gigahertz für die Schwingung des TB. Um diese Verzögerung des TB unabhängig von den Spezifikationen des verfügbaren Oszilloskops messen zu können, ist daher ein asynchroner Frequenzteiler erforderlich, um die Frequenz der Oszillation vor der Messung herunterzurechnen. Der verwendete N-stufige asynchrone Frequenzteiler ist in Abbildung 3.10 dargestellt. Er besteht aus einem LUT-basierten Datenlatch (D-Latch) und einem N-1-stufigen asynchronen Zähler. Das D-Latch arbeitet als einstu-



(a) GL Repräsentation der Messschaltung

(b) Symbol

Abbildung 3.10: N-stufiger Asynchroner Frequenzteiler

figer Frequenzteiler. Dies hat sich als notwendig erwiesen, da der nachfolgende Zähler sonst nicht in der Lage wäre, jede einzelne positive Flanke zu erkennen, insbesondere bei Oszillationen des TB mit relativ hohen Frequenzen. Damit das LUT-basierte D-Latch jedoch als asynchroner Frequenzteiler funktioniert, muss das Datensignal dem aktuellen Zustand des Latches entsprechen, aber umgeschaltet werden, und die FLPD des Latches muss mit der FLPD des TB übereinstimmen. Um die Gleichheit der Verzö-

gerungen zu gewährleisten, werden die gleichen LUTs verschiedener CLB's verwendet und auf die gleiche Weise verschaltet, was zu einem gleichwertigen Routing der Drähte im FPGA und gleichen Verzögerungsschätzungen von Vivado [77] führt.

Der asynchrone Zähler wird als Ripple-Through-Zähler [78] unter Verwendung der verfügbaren D-FFs im FPGA realisiert. Indem der Dateneingang D auf eine logische 1 gesetzt wird und der asynchrone Reset-Eingang zur Erzeugung der logischen 0 verwendet wird, wird der zusätzliche Inverter der klassischen Struktur mit D-FF eingespart, was zu einer wesentlich höheren maximalen Schaltfrequenz des Zählers führt [79].

## Messung

Der TB wurde zusammen mit dem 10-stufigen Frequenzteiler im FPGA implementiert und mit Vivado 2020.2 programmiert. Die Quellcodes wurden in Verilog geschrieben, um zu demonstrieren, dass sowohl VHDL als auch Verilog für den asynchronen Entwurf geeignet sind. Das Messbeispiel besteht aus 16 LUTs, die den vier Slices X0Y0, X1Y0, X2Y0 und X3Y0 des FPGAs zugeordnet wurden. Für jeden Eingang einer LUT wurden zehn Messungen durchgeführt, so dass sich insgesamt 960 Messungen ergeben. Während des gesamten Experiments wurde die Chiptemperatur auf  $(31,7 \pm 0,6)^\circ\text{C}$  gehalten. Das Signal wurde mit dem Oszilloskop Tektronix TDS1012B mit einer Aufzeichnungslänge von 2500 pts und einer Abtastrate von bis zu 1 GS/s für periodische Signale aufgezeichnet. Wenn bedacht wird, dass der 10-stufige Frequenzteiler das schwingende Signal um einen Faktor von  $2^{-10} \approx 10^{-3}$  auf mehrere Megahertz skaliert, wurde ein Zeitintervall von  $5\mu\text{s}$  in Kombination mit einer Abtastrate von 500 MS/s als angemessener Kompromiss zwischen der Anzahl der Bilevel-Pulse pro Aufzeichnung und der Auflösung eines einzelnen Pulses gefunden. Abbildung 3.11 zeigt das gemessene Oszillationssignal für einen in LUT A von Slice X0Y0 implementierten TB unter Verwendung des Eingangs A5 für das Rückkopplungssignal. Zur Berechnung der FLPD wurde zunächst der Mittelwert der Breiten aller über zehn Instanzen gemessenen Bilevel-Pulse ermittelt und dann um den Faktor  $2^{-10}$  herunterskaliert, um den 10-stufigen Frequenzteiler zu kompensieren. Der Vollständigkeit halber wurde auch die korrigierte Stichprobenstandardabweichung berechnet. Die verallgemeinerten Formeln sind in Gl.

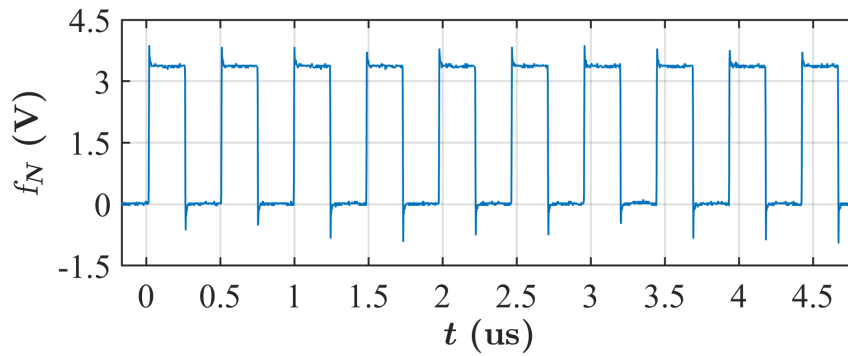


Abbildung 3.11: Oszillierendes Signal für einen TB in LUT A des Slices X0Y0 unter Verwendung von Eingang A5 als Rückkopplungspfad.

3.2 und Gl. 3.3 angegeben [80].

$$\tau_{FLPD,avg} = \frac{1}{M(T-1)} \sum_{i=1}^M \sum_{j=1}^{T-1} \frac{t_{i,j+1} - t_{i,j}}{2^N} \quad (3.2)$$

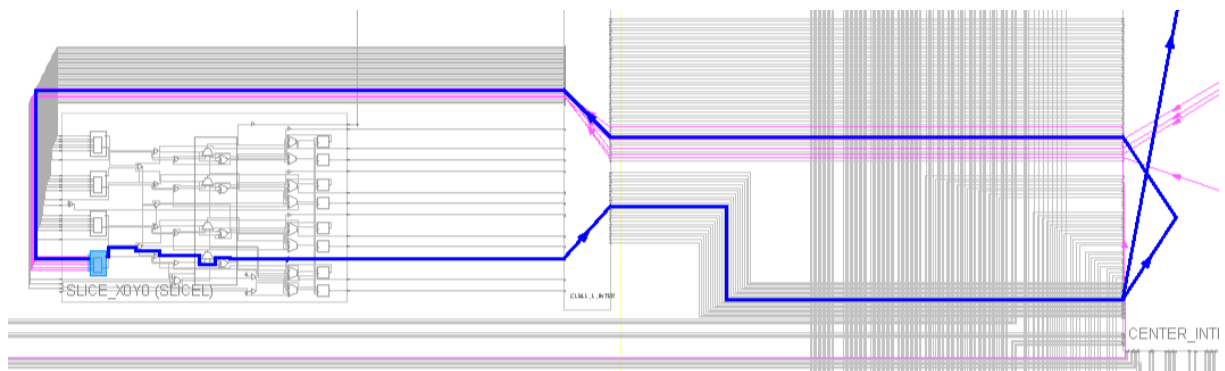
$$\sigma_{FLPD}^2 = \frac{1}{M(T-1) - 1} \sum_{i=1}^M \sum_{j=1}^{T-1} \left( \frac{t_{i,j+1} - t_{i,j}}{2^N} - \tau_{FLPD,avg} \right)^2 \quad (3.3)$$

Der Parameter  $N$  enthält die Anzahl der Stufen des Frequenzteilers,  $M$  steht für die Anzahl der durchgeführten Messungen und der Parameter  $T$  markiert die Anzahl der Übergänge des aufgezeichneten Signals bei 1.65 V als mittlerem Spannungspegel zwischen einer logischen 0 und einer logischen 1. Die Zeitvariablen  $t_{i,j+1}$  und  $t_{i,j}$  tragen die linear interpolierten Zeitwerte für zwei aufeinander folgende Übergänge bei 1,65 V des  $i$ -ten Messsignals. Die Ergebnisse der Berechnung des Mittelwerts und der Standardabweichung des FLPD für das in Abbildung 3.11 angegebene Signal und die Signale, die bei Verwendung der anderen Eingänge der LUT für das Rückkopplungssignal erhalten wurden, sind in Tabelle 3.2 gegeben. Offensichtlich ist das FLPD minimal, wenn

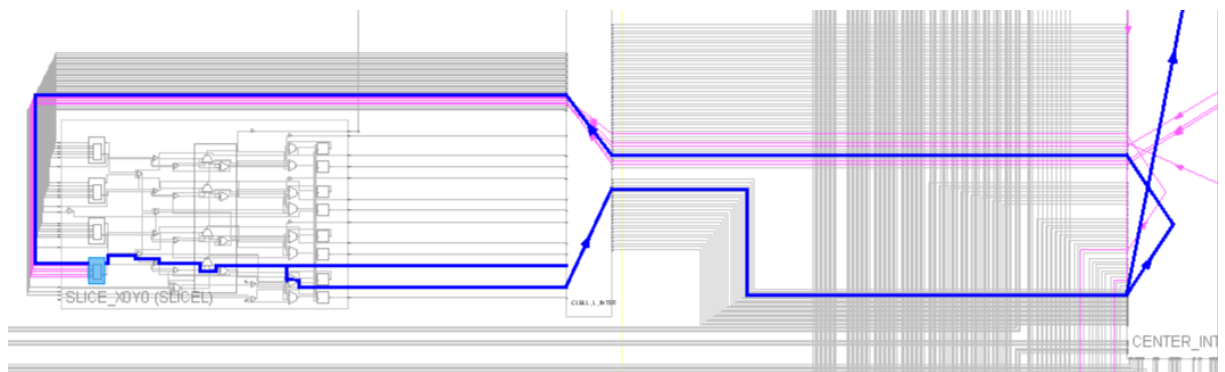
Tabelle 3.2: Mittlere FLPD des TB in LUT A des Slices X0Y0

	A1	A2	A3	A4	A5	A6	
$\tau_{FLPD,avg}$	595.1	582.2	477.9	435.8	239.2	290.3	[ps]
$\sigma_{FLPD}$	0.6	0.6	0.5	0.6	0.5	0.5	[ps]

Eingang A5 für das Rückkopplungssignal verwendet wird und nicht Eingang A6, wie die Struktur des TB in Abbildung 3.8 vermuten ließ. Der Grund dafür ist der Unterschied im Routing der beiden Implementierungen, wie der Vergleich von Abbildung 3.12(a) und Abbildung 3.12(b) zeigt. Abbildung 3.13 zeigt die durchschnittlichen FLPD-Ergebnisse



(a) Input A5



(b) Input A6

Abbildung 3.12: Implementierung des TB in LUT A des Slices X0Y0 unter Verwendung von A5 oder A6 als Rückkoppelpfad (highlighted)

für alle 16 LUTs. Es wurde nur zwischen den LUTs, nicht aber zwischen den Slices unterschieden, da für verschiedene Slices ähnliche Ergebnisse erwartet wurden. Die Standardabweichung ist nicht dargestellt, da sie zu klein ist, um auf der y-Achse angezeigt zu werden. Die Breite der horizontalen Linien in der Abbildung hat keine Bedeutung und dient nur der besseren Lesbarkeit. Der minimale gemessene FLPD betrug  $220,7 \text{ ps}$  mit einer Standardabweichung von  $0,6 \text{ ps}$  für LUT D in Slice X1Y0 unter Verwendung von Input A5. Der maximale gemessene FLPD betrug  $759,6 \text{ ps}$  mit einer Standardabweichung von  $0,6 \text{ ps}$  für LUT B in Slice X1Y0 unter Verwendung von Eingang A2. Hinsichtlich der minimalen FLPD stimmen die Ergebnisse der verschiedenen Slices nur mit Tab. 3.2 für die LUTs A und D überein, aber nicht für die LUTs B und C. Auch hier ist der Unterschied im Routing die Ursache. Zur Vervollständigung: Tab. 3.3 listet die minimale FLPD für jede der 16 LUTs auf. Für die weitere Betrachtung des Muller C wird die LUT A des Slice X0Y0 verwendet, wobei die Rückkopplungsleitung mit dem Eingang A5 verbunden ist, da diese Konfiguration ein minimales FLPD ergibt. Die genaue Struktur ist in Abbildung 3.14 dargestellt, wobei das FLPD die Summe der

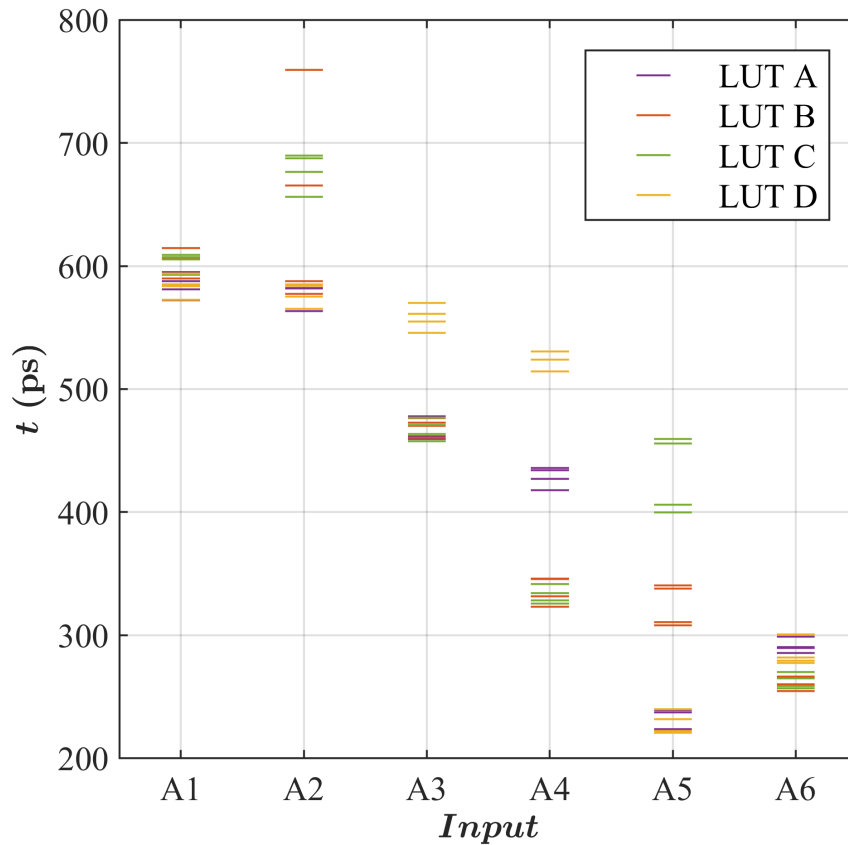


Abbildung 3.13: FLPDs der TB implementiert in LUT A-D von Slice X0Y0-X3Y0

Slice	$\tau_{FLPD,avg}$	$\sigma_{FLPD}$
X0Y0	239.2	0.5
X1Y0	221.0	0.5
X2Y0	237.2	0.5
X3Y0	223.5	0.5

(a) Input A5, LUT A

Slice	$\tau_{FLPD,avg}$	$\sigma_{FLPD}$
X0Y0	260.3	0.6
X1Y0	254.6	0.6
X2Y0	266.3	0.6
X3Y0	254.6	0.6

(b) Input A6, LUT B

Slice	$\tau_{FLPD,avg}$	$\sigma_{FLPD}$
X0Y0	264.8	0.5
X1Y0	256.7	0.6
X2Y0	270.0	0.6
X3Y0	258.6	0.5

(c) Input A6, LUT C

Slice	$\tau_{FLPD,avg}$	$\sigma_{FLPD}$
X0Y0	231,7	0.5
X1Y0	220.7	0.6
X2Y0	239.9	0.5
X3Y0	222.2	0.9

(d) Input A5, LUT D

Tabelle 3.3: Minimum FLPD der LUTs A-D der Slices X0Y0-X3Y0

Vorwärtspfadverzögerung  $\tau_\delta$  und der Rückwärtspfadverzögerung  $\tau_\Delta$  ist.

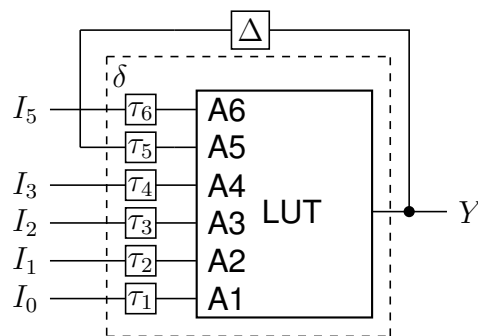


Abbildung 3.14: Rückkopplung auf eine sechs Eingänge umfassende LUT abgebildet

## Test

Um sicherzustellen, dass die Beziehung  $\tau_\Delta < \tau_\delta$  gültig ist, wird ein Fehlermodell aufgestellt, welches das Verhalten im Fehlerfall ( $\tau_\Delta > \tau_\delta$ ) darstellt. Auf diesen Fehler wird getestet. Für den Test wird eine Impulsschaltung entworfen, die beim Schalten des Eingangs  $i$  einen kurzen Impuls der Länge  $\tau \geq \tau_\delta$  erzeugt. Wenn  $\tau_\Delta > \tau_\delta$  ist, muss gewährleistet sein, dass die Beziehung  $\tau \leq \tau_\Delta$  gilt, sonst kann der Fehler am Ausgang nicht erkannt werden. Der Impuls ist dann der Eingang für eine rückgekoppelte ODER-Schaltung, siehe Abbildung 3.15. Das Fehlermodell besagt, dass der Fehler

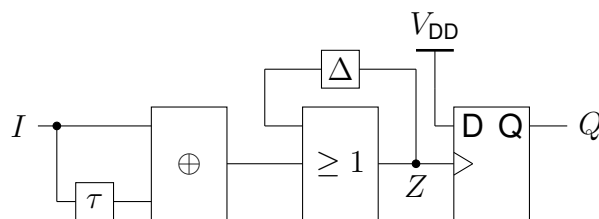


Abbildung 3.15: Testschaltung zur Verifikation

und die Beziehung  $\tau_\Delta > \tau_\delta$  gültig sind, wenn die OR-Schaltung nicht auf 1 gesetzt wird, aber einen Impuls durchgelassen hat. Um sicherzustellen, dass tatsächlich ein Impuls durchgelassen wurde, wird der Clk-Eingang eines DFF mit dem Ausgang der ODER-Schaltung verbunden. Der D-Eingang wird auf  $V_{DD}$  gesetzt, so dass der D-FF beim Auftreten eines Impulses eine 1 speichert, so dass im Fehlerfall nach einer bestimmten Zeit  $t_1$  eine 0 bei  $z$  und eine 1 im DFF anliegt. Wenn die Zuordnung [11] an

den Ausgängen ( $z, q$ ) sichtbar ist, wurde ein funktionsstabiles Muller-C-Element implementiert. Die erwarteten Signale für das Fehlermodell sind in Abbildung 3.16 zu sehen, wobei  $\tau = \tau_\delta$  gilt.

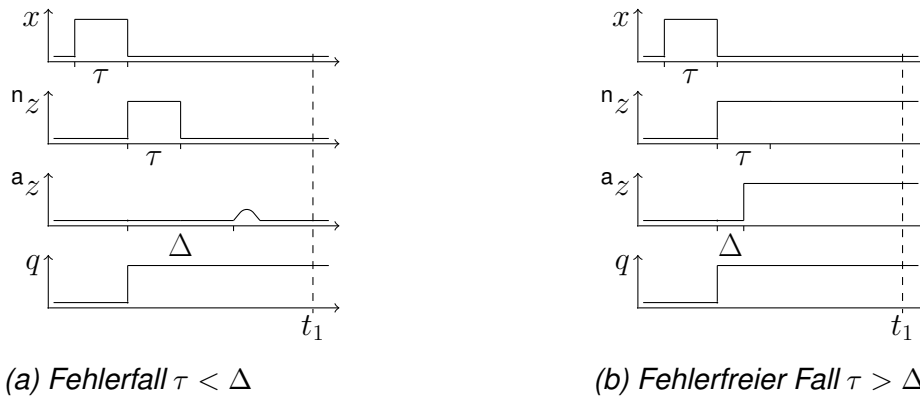


Abbildung 3.16: Testfälle

Die Test lautet nun, wenn  $q = 1$  gilt:

$${}^n z(t_1) = 0 \quad \text{fail} \quad (3.4)$$

$${}^n z(t_1) \neq 0 \quad \text{pass} \quad (3.5)$$

und besagt, dass die Fehlerbedingung vorliegt, wenn der Ausgang  $z = 0$  ist. Eine 0 am Ausgang zeigt also eindeutig an, dass der Rückwärtspfad langsamer ist als der Vorwärtspfad und es keine funktionsstabile Konstruktion gibt. Die Ergebnisse des Tests zeigen, dass bei Verwendung von Dateneingang A6 als Dateneingang der Fehler auftritt und das Müller-C-Element nicht funktionsstabil ist. Werden dagegen die Dateneingänge A4 bis A1 verwendet, tritt der Fehler nicht mehr auf, und es wird logisch geschlossen, dass die Schaltung funktionsstabil ist. Daher werden für das schnellste und sicherste Müller-C-Element die schnelleren Dateneingänge A4 und A3 bevorzugt, siehe Abbildung 3.17, wobei die nicht verwendeten Eingangspins mit  $V_{DD}$  verbunden sind. Mit diesem funktionsstabilen Muller-C-Element können nun sichere asynchrone Schaltungen in FPGAs implementiert werden. Die Vivado Design Suite hat sich dabei als gut zu verwendendes Tool für die manuelle Anpassung von asynchronen Schaltungen erwiesen.

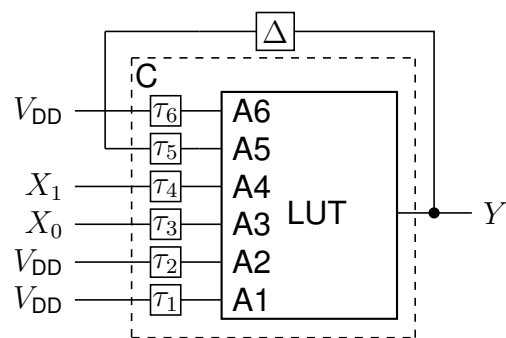


Abbildung 3.17: Funktionsstabiles Muller-C-Element in einer sechs Eingängen umfassenden LUT realisiert

## 4 Entwurfsmethoden Asynchroner Schaltungen

Als Vergleich werden im folgenden Kapitel zwei Entwurfsmethoden von asynchronen Schaltungen im FPGA vorgestellt. Dabei handelt es sich zum einen um das einschrittige Automaten-Design, das asynchron im Sinne von gänzlich taktlos versteht und delay-insensitive ist und zum anderen das asynchrone Design mithilfe von Dominologikgattern, die selbstsperrend entworfen sind, das asynchron im Sinne von Handshaking-Protokollen und fehlendem globalen Takt versteht und eine self-timed Schaltung darstellt. Zunächst werden die Entwurfsmethoden theoretisch aufgezeigt, dann werden Beispielschaltungen realisiert. Die Realisierungen werden verglichen in Bezug auf Komplexität, Sicherheit und Geschwindigkeit.

### 4.1 Einschrittiges Automaten-Design

Zunächst soll eine Entwurfsmethode vorgeschlagen werden, wie Automaten ohne Takt stabil realisiert werden können. Beim einschrittigen Automaten dreht sich prinzipiell alles um einschrittige Änderung von Variablen (Eingangs- und z-Variablen). Das wird in [81] fundamental mode genannt. Zunächst sollen die Eingangsvariablen näher betrachtet werden.

#### 4.1.1 Einschrittige Kodierung der Eingangsvariablen

Jeder Zustandswechsel eines Automaten  $A$  soll nur durch Wechsel von einer Variablen im Eingang möglich sein. Die Änderung des Zustandes muss dann in einer Eigenschleife enden und muss damit funktionsstabil sein. Der Automat ist damit einschrittig in den  $x$ -Variablen und einschrittig in den Zustandsübergängen. Dies wird durch den partiellen Automaten in Abbildung 4.1 dargestellt. Die Darstellung als Phasenliste (Wertetabelle) findet sich in Tabelle 4.1, die Darstellung als Multiset (KV-Diagramm) in Abbildung 4.2. Daraus lassen sich die unären  $z$ -Variablen  $z_1 = Z_1^1$  und  $z_0 = Z_0^1$  aus den logischen Zustandsfunktionen  $\delta_z = (\delta_{z_1}, \delta_{z_0})$  von Gleichung 4.1 bis Gleichung 4.2

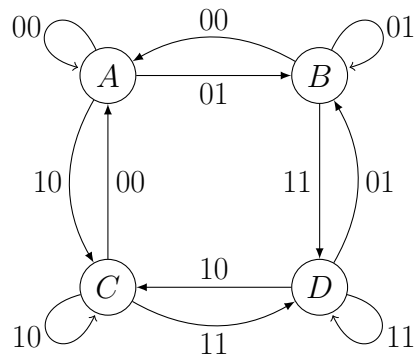


Abbildung 4.1: Einschrittig x-kodierter Automat (Graph)

$i$	$z_1$	$z_0$	$x_1$	$x_0$	$z_1$	$z_0$
0	0	0	0	0	0	0
1	0	0	0	1	0	1
2	0	0	1	0	1	0
3	0	0	1	1	*	*
4	0	1	0	0	0	0
5	0	1	0	1	0	1
6	0	1	1	0	*	*
7	0	1	1	1	1	1
8	1	0	0	0	0	0
9	1	0	0	1	*	*
A	1	0	1	0	1	0
B	1	0	1	1	1	1
C	1	1	0	0	*	*
D	1	1	0	1	0	1
E	1	1	1	0	1	0
F	1	1	1	1	1	1

Tabelle 4.1: Phasenliste des Automaten

bestimmen:

$$\delta_{z_1}(z, x) = z_1x_1 \vee \bar{z}_0x_1\bar{x}_0 \vee z_0x_1x_0 \quad (4.1)$$

$$\delta_{z_0}(z, x) = z_0x_0 \vee z_1x_1z_0 \vee \bar{z}_1\bar{x}_1x_0 \quad (4.2)$$

Bei einer zweistufigen Implementierung von Gattern sind keine Funktions hazards mehr vorhanden. Während der Übergänge ändern sich die Signale jeweils nur in einer Variablen. Sterne (\*) sind verbotene Zuweisungen [10] und durch Don't Cares in Abbildung 4.3 implementiert.

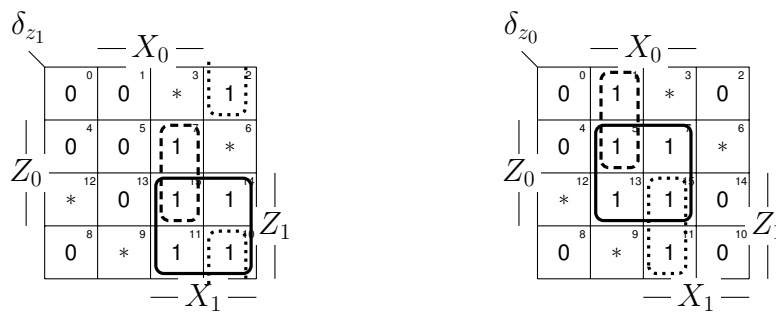
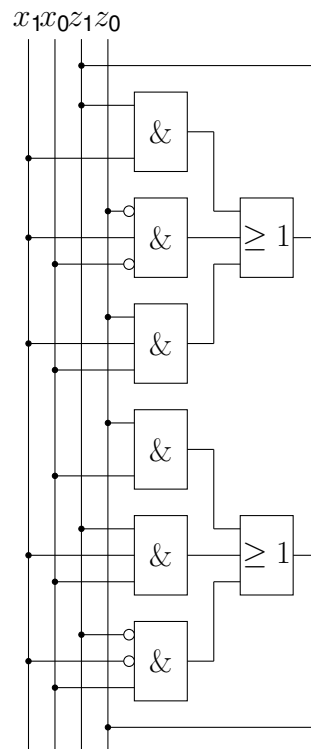
Abbildung 4.2: KV-Diagramm von  $\delta_{z_1}$  und  $\delta_{z_0}$ 

Abbildung 4.3: Digitale Schaltung des Automaten

#### 4.1.2 Einschrittigkeit der z-Variablen

Werden nur einschrittig ändernde Eingangsbelegungen zugelassen, kann sich nur ein Pfad ändern und somit ist der Vorwärtspfad sicher. Bei einem Automaten kann es aber auch zu Änderungen im Rückwärtspfad kommen und somit keine Einschrittigkeit gewährleistet sein, es können mögliche Races auftreten.

Nun müssen noch die Zustände belgt werden, dass nur einschrittige Übergänge entstehen [82]. In diesem Beispiel ist es einfach die geeignete Kodierung zu finden, da jeder Zustand maximal zwei Zustandsverbindungen hat und somit im 2-dimensionalen

Kodierungsuniversum  $(z_1, z_0)$  jeweils eine Stelle sich ändern kann. Der kodierte Automat ist in Abbildung 4.4 zu sehen. Abbildung 4.1 dargestellt. Würde die

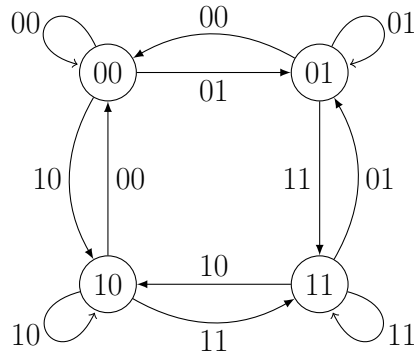


Abbildung 4.4: Einschrittig kodierter Automat (Graph)

Kante [11] vom Zustand  $Z_0$  in den Zustand  $Z_1$  führen, siehe Abbildung 4.5, ist das Beispiel etwas schwieriger zu kodieren. Da  $Z_0$  nun drei Zustandsübergänge besitzt, muss

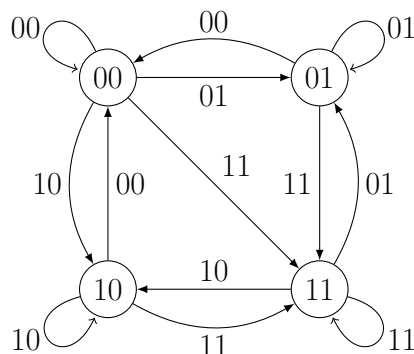


Abbildung 4.5: Mehrschrittiger Automat (Graph)

der umkodierte Automat drei  $z$ -Variablen enthalten, um einschrittig realisiert zu werden. Zudem ist es erforderlich, teilweise mit transitiven Zuständen zu arbeiten, da der Übergang von  $Z_1$  nach  $Z_3$  über einen zusätzlichen transitiven Zustand erfolgen muss, um die Einschrittigkeit sicherzustellen. Es existieren Algorithmen zur einschrittigen Kodierung, siehe z.B. [82] oder [83], jedoch wird diese Thematik im weiteren Verlauf der Arbeit nicht weiter behandelt.

### 4.1.3 Perfekter Automat

Ein funktionshazardfreier Automat, siehe Abbildung 4.6, zur Richtungserkennung, inspiriert durch [84], wurde entwickelt. Dabei wurde jedoch eine alternative Lösung erarbeitet. Es ist ein Medwedew-Automat zu sehen, der die Eingänge  $X_1$  und  $X_0$  und die

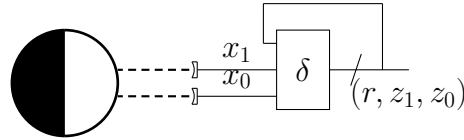


Abbildung 4.6: Richtungserkennung eines drehenden Objekts

$z$ -Variablen  $r, z_1, z_0$ . Die Eingänge sind dabei Reflexionssensoren, die messen, ob das Signal reflektiert oder absorbiert wird, also ob die weiße oder schwarze Oberfläche der sich drehenden Kreisscheibe zu sehen ist. Die  $z$ -Variablen  $z_1$  und  $z_0$  geben dabei den alten Zustand der Kreisscheibe wieder und die Eingänge den neuen Zustand. Bei jedem Übergang kann sich je nur eine einzige  $x$ -Variable ändern, da die Abtastung um vieles schneller als die Drehung der Scheibe geschieht. Dadurch, dass  $z$ -Variable den neuen Zustand, also den neuen Wert von  $x$  übernimmt, können sich auch die  $z$ -Variablen nur einschrittig ändern. Dies wird als *correct-by-construction* bezeichnet.

Da bei Richtungswechseln sich auch die  $r$ -Variable ändern kann, wird der Automat geschickt umkodiert, sodass die race-freie Implementierung der Schaltung in Abbildung 4.7 herauskommt. Um eine Glitch-freie Schaltung zu realisieren, werden die ungenutzten Flanken, die nicht auftreten können, als *hold* kodiert, d. h. der Automat behält den letzten Zustand bei. Mit Hilfe der KV-Diagramme der unären Variablen  $R, \bar{R}, Z_1, \bar{Z}_1, Z_0$  und  $\bar{Z}_0$ , siehe Abbildung 4.8 bis Abbildung 4.10, können die Zustandsübertragungsfunktionen in Gleichung 4.3 bis Gleichung 4.8 aufgestellt werden.

$$r = rz_1x_0 \vee rz_0\bar{x}_1 \vee r\bar{z}_1\bar{x}_0 \vee r\bar{z}_0x_1 \vee \bar{z}_1z_0\bar{x}_1x_0 \vee z_1z_0x_1x_0 \vee z_1\bar{z}_0x_1\bar{x}_0 \vee \bar{z}_1\bar{z}_0\bar{x}_1\bar{x}_0 \quad (4.3)$$

$$\bar{r} = \bar{r}z_1\bar{x}_1 \vee \bar{r}\bar{z}_1x_1 \vee \bar{r}\bar{z}_0x_0 \vee \bar{r}z_0\bar{x}_0 \vee z_1z_0x_1\bar{x}_0 \vee \bar{z}_1z_0x_1x_0 \vee z_1\bar{z}_0\bar{x}_1\bar{x}_0 \vee \bar{z}_1\bar{z}_0\bar{x}_1x_0 \quad (4.4)$$

$$z_1 = z_1x_1 \vee z_1\bar{x}_0 \vee \bar{r}z_1z_0 \vee rz_1\bar{z}_0 \vee \bar{r}z_0x_1\bar{x}_0 \vee r\bar{z}_0x_1\bar{x}_0 \quad (4.5)$$

$$\bar{z}_1 = \bar{z}_1\bar{x}_1 \vee \bar{z}_1x_0 \vee r\bar{z}_1z_0 \vee \bar{r}\bar{z}_1\bar{z}_0 \vee rz_0\bar{x}_1x_0 \vee \bar{r}\bar{z}_0\bar{x}_1x_0 \quad (4.6)$$

$$z_0 = z_0x_1 \vee z_0x_0 \vee rz_1z_0 \vee \bar{r}\bar{z}_1z_0 \vee rz_1x_1x_0 \vee \bar{r}\bar{z}_1x_1x_0 \quad (4.7)$$

$$\bar{z}_0 = \bar{z}_0\bar{x}_1 \vee \bar{z}_0\bar{x}_0 \vee r\bar{z}_1\bar{z}_0 \vee \bar{r}z_1\bar{z}_0 \vee r\bar{z}_1\bar{x}_1\bar{x}_0 \vee \bar{r}z_1\bar{x}_1\bar{x}_0 \quad (4.8)$$

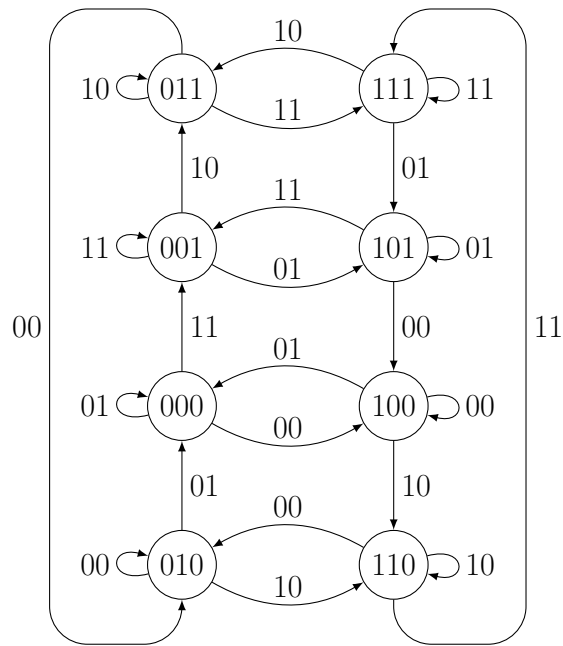


Abbildung 4.7: Einschrittig kodierter Rotationserkennungsautomat

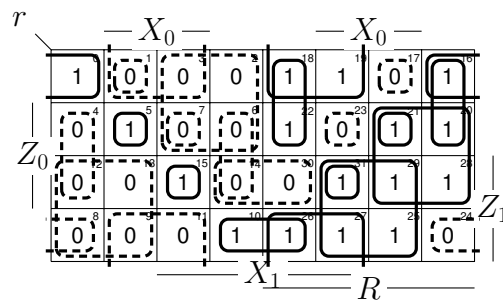


Abbildung 4.8: KV-Diagramm von  $r = (R, \bar{R})$

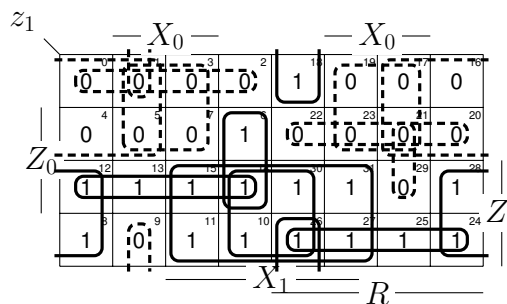


Abbildung 4.9: KV-Diagramm von  $z_1 = (Z_1, \bar{Z}_1)$

Die sich daraus ergebende Dual-Rail-Implementierung des Automaten ist in Abbildung 4.11 gegeben.

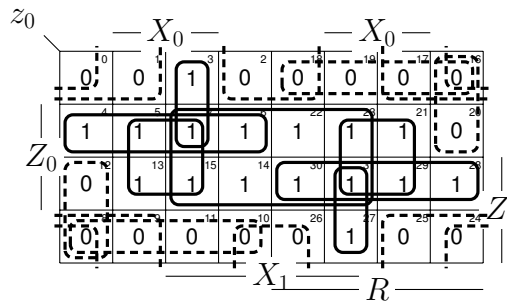


Abbildung 4.10: KV-Diagramm von  $z_0 = (Z_0, \bar{Z}_0)$

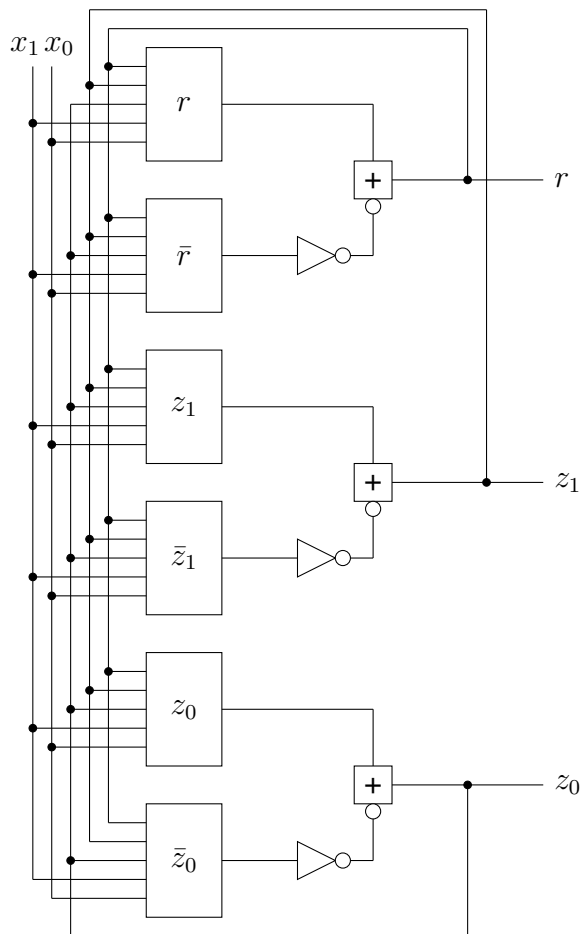


Abbildung 4.11: Glitch-freie Dual-Rail-Implementierung

#### 4.1.4 Fazit zum taktlosen Automaten

Der komplett taktlose asynchrone Automat ist prinzipiell möglich zu realisieren, hat aber sehr viele Einschränkungen, wie z.B. die einschränkte Eingangs- und z-Variablen-

Änderung und die Hazardfreiheit durch überlappende Blöcke zweistufiger Funktionen. Während einige spezielle Schaltungen so realisiert werden können, kann i.A. nicht davon ausgegangen werden, dass sich z.B. die Eingänge nur einschrittig ändern dürfen. Es wäre möglich Schaltungen dazwischenzuschalten, um Einschrittigkeit zu erzwingen, das verschiebt jedoch das Problem bzw. den Entwurfsaufwand Richtung Eingang. Zudem müssen beim taktlosen Automaten die Zustände zwingend funktionsstabil entworfen werden, da sonst einige Zustände so transitiv werden, dass sie nicht mehr als Zustand angesehen werden können, daher entsteht der Bedarf einer allgemeiner anwendbaren asynchronen Entwurfsmethodik.

## 4.2 Selbstsperrende Dual-Rail-Dominologik

Ziel ist es, eine Entwurfsmethode zu konzipieren, die wenig Einschränkungen erfordert und sicher ist. Hierfür wird in diesem Kapitel gezeigt, wie selbstsperrende DRDL entworfen wird, die durch Selbsttaktung und Selbstsperrung und die Bündelung der Eingangsdaten beliebige Eingangsänderungen erlaubt, Race-frei durch den Domino-Effekt ist und durch die Dual-Rail-Implementierung auch gegen Glitches gehärtet ist. In diesem Abschnitt wird die Umsetzung der selbstsperrenden Dominologik vorgestellt. Die Schaltung basiert auf einer Pulsschaltung am Eingang, die die gesamte Schaltung sperrt, bis eine vollständige Transition abgeschlossen ist und der Dominologikschaltung, die die Pulsschaltung anschließend wieder entsperrt, wenn alle Signal-Übergänge passiert sind. Dadurch taktet sich die Schaltung selbst und signalisiert, sobald die eingehenden Daten vollständig verarbeitet wurden.

Zunächst wird ein Strukturvergleich zwischen LUTs und Dominogattern durchgeführt, um zu analysieren, wie sich Dominologik auf einem FPGA realisieren lässt. Anschließend wird die selbstsperrende Pulsschaltung entwickelt, die nach der Übernahme der gebündelten Daten den Eingang sperrt und einen Dutycycle erzeugt.

Dieser Dutycycle versetzt die Dominogatter in ihre Precharge-Phase, wodurch ein invalides Datensignal (Spacer) generiert wird. Anschließend erfolgt die Evaluierungsphase, in der die Dominogatter schalten, bis alle DRDL-Gatter einen disjunkten Ausgang aufweisen. Die Low-Level-Realisierung der DRDL-Gatter wird daraufhin gezeigt. Die disjunkten Ausgänge zeigen dann über eine XOR an, wenn die Ausgangssignale valide sind, woraufhin der Schaltkreis wieder entsperrt, sodass neue Daten verarbeitet werden können.

### 4.2.1 Single-Rail-Dominologik im FPGA

Zunächst werden wir eine SRDL-Schaltung mit einem Keeper auf TL, wie in Abbildung 4.12 gezeigt, betrachten. Das wird nun durch eine MUX-Struktur realisiert, welche

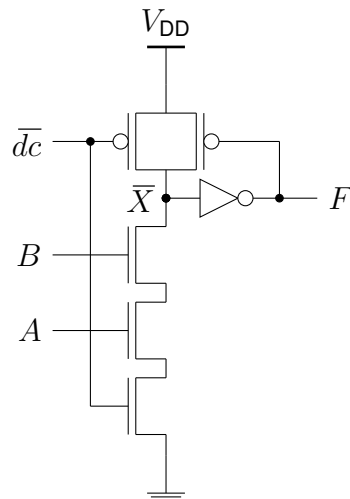


Abbildung 4.12: Single-Rail Dominologik

aus Pass-Transistoren besteht. Es handelt sich um eine LUT3 siehe Abbildung 4.13. Da alle vier unteren Pfade für  $\overline{dc} = 0$   $V_{DD}$  treiben (Precharge), wurden diese der Ein-

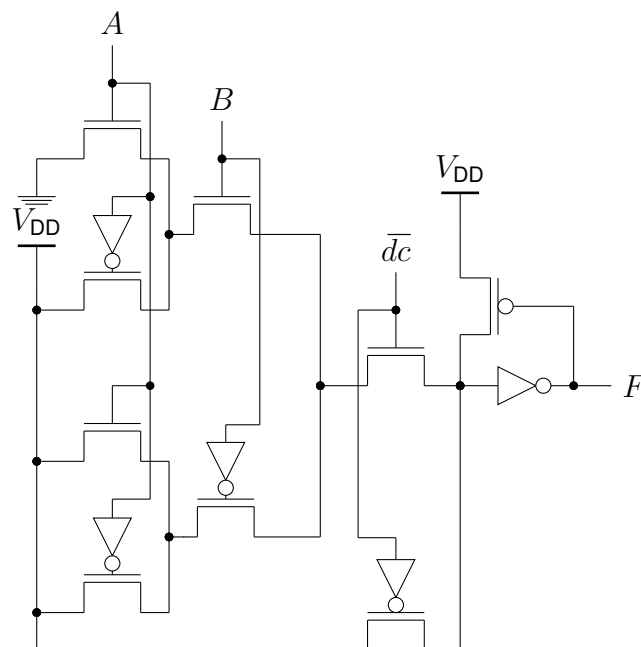


Abbildung 4.13: SRDL realisiert durch eine LUT3

fachheit halber weggelassen und ein direkter Pfad von  $V_{DD}$  über  $\overline{dc}$  gezeichnet. Der

Knoten vor dem NMOS, der von  $\bar{dc}$  gesteuert wird, kann nur bis  $V_{DD}$  geladen werden oder auf High-Z gehen und die Ladung halten. Daher gibt diese Vereinfachung die Struktur genau wieder. Außerdem kann der Knoten  $\bar{X}$  nur über  $AB$  nach GND gezogen werden, was bedeutet, dass der obere Pfad der einzige ist, der den Übergang von 1 nach 0 auslösen kann. Der untere Pfad lädt also von 0 nach 1 (Precharge), und der obere Pfad entlädt sich über  $AB$  von 1 nach 0 (Evaluate). Indem wir diese Vereinfachung umsetzen und nur die Pfade für die Übergänge darstellen, erhalten wir die in Abbildung 4.13 dargestellte Struktur mit der Ausnahme, dass der Transistor für Evaluate näher am Ausgang liegt, wie in Abbildung 4.14 dargestellt. Die Struktur kann

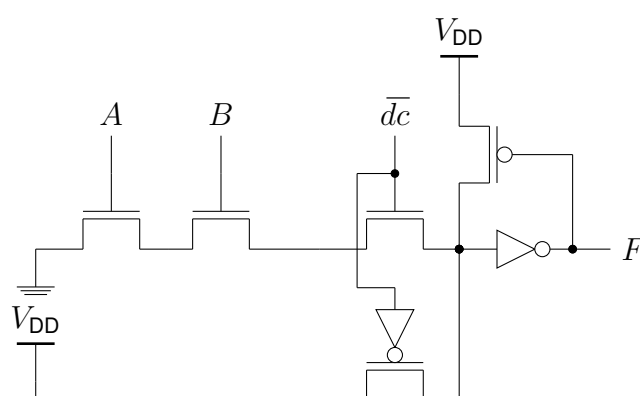


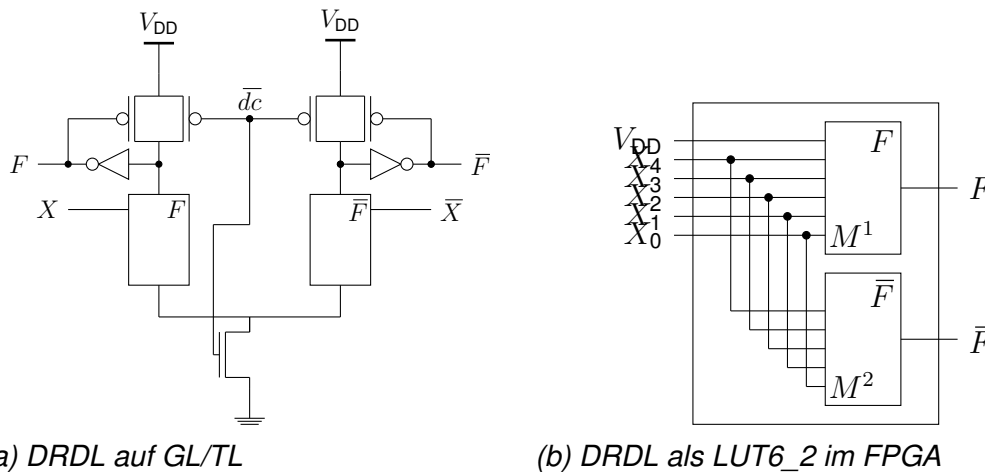
Abbildung 4.14: Transitionen von SRDL abgebildet auf eine LUT3

vergleichbar aufgebaut werden, wenn die Kontrollsignale in anderer Reihenfolge verschaltet werden (also  $[dc]$  auf den Input, der am weitesten vom Ausgang entfernt ist).

#### 4.2.2 Dual-Rail-Dominologik

Ein DRDL-Gatter mit maximal vier Dateneingängen kann im Artix-7 in einer einzigen LUT realisiert werden, wie in Abbildung 4.15 zu sehen ist.

Die LUT6\_2 wird so beschaltet ( $I_5$  auf  $V_{DD}$ ), dass beide Ausgänge disjunkt zueinander sind. Dies geschieht, indem durch die 1 auf  $I_5$  für  $O6$  der obere Pfad ausgewählt wird. Dadurch trennen sich die Belegungen der Tabelle komplett in den Belegungen 0 bis 31 für  $O5$  und den Belegungen 32 bis 63 für  $O6$ . Dadurch entwerfen wir die zwei disjunkten Ausgänge  $F$  und  $\bar{F}$ . Beide werden vom selben Signal  $\bar{dc}$  gesteuert. Da beide Funktionen komplementär zueinander entworfen werden, wird in der Evaluierungs-Phase das valide Datensignal  $[01]$  oder  $[10]$  für  $(F, \bar{F})$  erzeugt. Somit ist klar ersichtlich, sobald die Schaltung durchgeschaltet hat und fertig ist. Der Code für ein DRDL-Gatter ist in Listing 4.1 gegeben.



(a) DRDL auf GL/TL

(b) DRDL als LUT6\_2 im FPGA

Abbildung 4.15: Dual-Rail-Dominologik

```

LUT6_2_inst : LUT6_2
generic map (
  INIT => X"800000007FFF0000") --
port map (
  06 => f_int, -- 6/5-LUT output (1-bit)
  05 => fbar_int, -- 5-LUT output (1-bit)
  I0 => '1', -- LUT input (1-bit)
  I1 => '1', -- LUT input (1-bit)
  I2 => x_int(0), -- LUT input (1-bit)
  I3 => x_int(1), -- LUT input (1-bit)
  I4 => _dc, -- LUT input (1-bit)
  I5 => '1'-- LUT input (1-bit)
);

```

Listing 4.1: Low-Level LUT6\_2 für das AND2 DRDL Gate

### 4.2.3 Selbstsperrung

An den Eingang des Dominologikmoduls wird nun eine Pulsschaltung angeschlossen, um eine Selbstsperrung am Eingang (self-X) zu realisieren, siehe Abbildung 4.17. Die self-X-Schaltung verriegelt sich selbst, wenn ein Impuls durchläuft, und sperrt den Eingang direkt. Sie wird erst wieder durch den *En*-Pin freigegeben, wenn die Daten in der Dominologikschaltung verarbeitet wurden. Wenn der Impuls nicht genügend Energie hat, wird kein Schaltvorgang ausgelöst. Um dieses Problem im FPGA zu lösen, wird eine funktionsstabile Schaltung verwendet. Diese Schaltung friert im internen Zustand bei 1 ein und setzt dann den Ausgang der Pulsschaltung auf HIGH (Evaluate), sobald

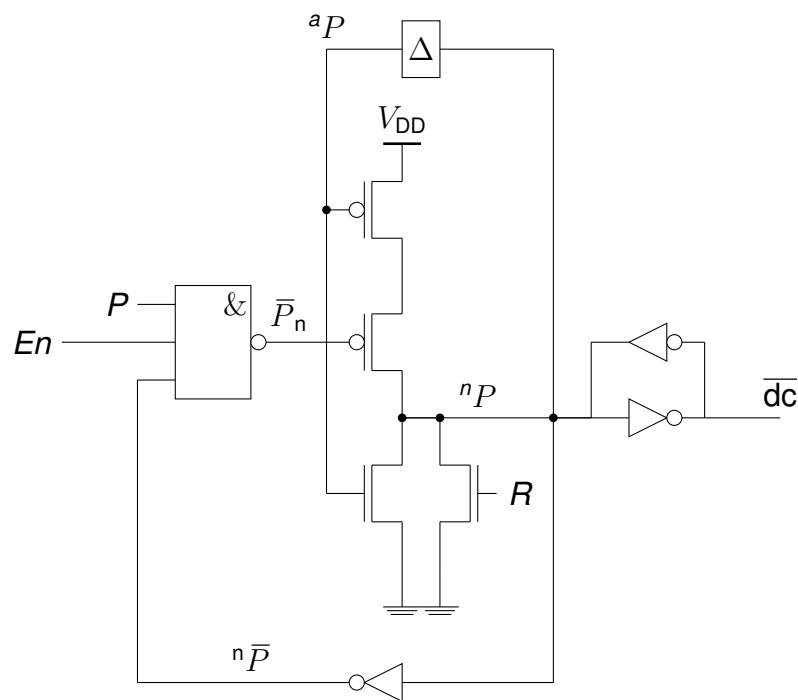


Abbildung 4.16: Pulsschaltung auf TL

eine Verzögerung von  $\Delta$  vergangen ist. Dies bleibt so lange der Fall, bis die Pipeline abgeschlossen ist, woraufhin der Eingang durch das Signal *en* wieder entsperrt wird.

Anstelle der Verwendung der grundlegenden FDCE-Komponente, eines DFF mit Clock Enable und asynchronem Reset, wie in [7] dargestellt, wird die entsprechende LUT-basierte Konfiguration aus Abbildung 4.17 eingesetzt.

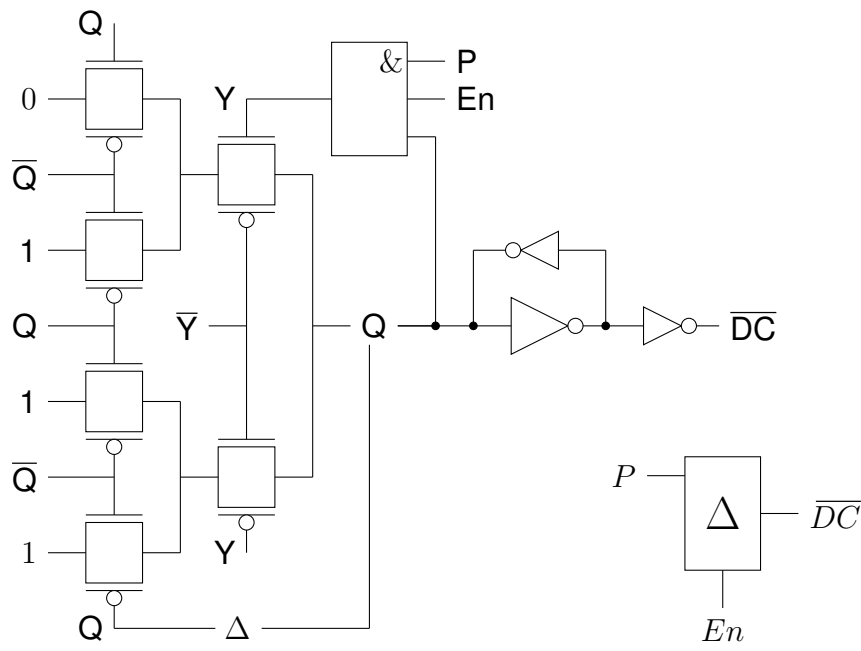
Die zugehörige Wahrheitstabelle befindet sich in Tabelle 4.2, und der Signalflussgraph ist in Abbildung 4.18 dargestellt.

Die Schaltung verriegelt sich selbst, sobald die Funktionsstabilität gewährleistet ist, was der Fall ist, wenn  $\tau_{\text{inv}} < \tau_{\text{LUT}}$  der LUT [1].

Tabelle 4.2: Wahrheitstabelle der Pulsschaltung

$\Delta(Q)$	Y	Q	Kommentar
0	0	1	Schalten
0	1	1	Schalten
1	0	1	Halten
1	1	0	Schalten

Die Gleichung für  $Q$  ist in Gleichung 4.9 angegeben.



(a) Self-resetting LUT-Struktur (b) Symbol (LL)

Abbildung 4.17: Pulsschaltung für Selbstsperrung und Duty Cycle

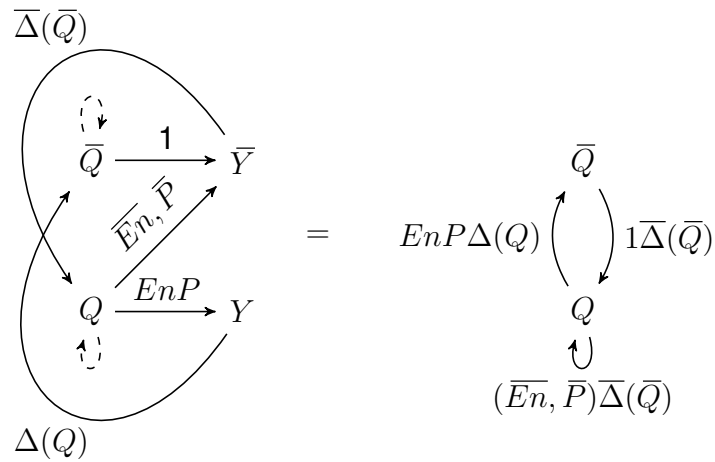


Abbildung 4.18: SFG der Pulsschaltung

$$Q \leftarrow \bar{Q}1\bar{\Delta}(\bar{Q}), Q(\bar{E}, \bar{P})\bar{\Delta}(\bar{Q}) \tag{4.9}$$

$$\bar{Q} \leftarrow QEnP\Delta(Q) \tag{4.10}$$

Das digitale Zeitdiagramm ist in Abbildung 4.19 dargestellt.

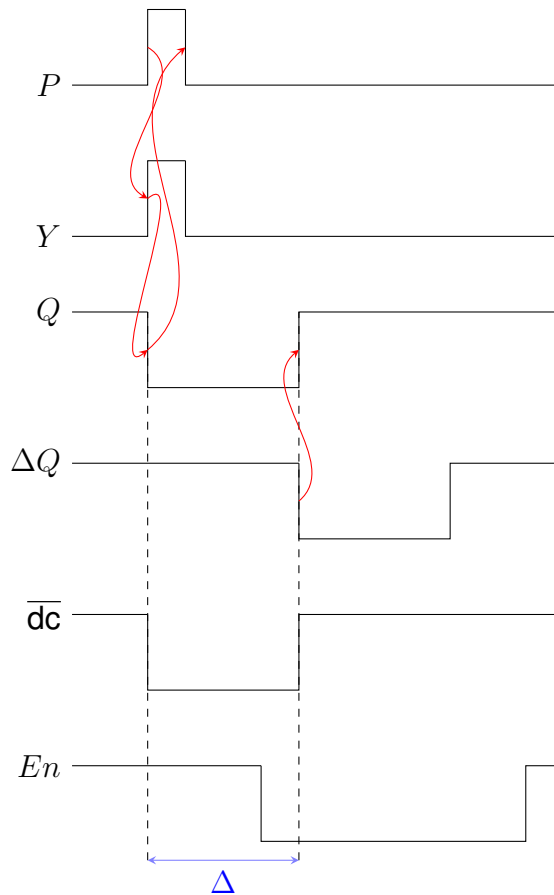


Abbildung 4.19: Digitales Impulsdiagramm

Die Low-Level-Implementierung in VHDL wird in Listing 4.2 angegeben.

```

attribute DONT_TOUCH of q_int : signal is "TRUE";
attribute ALLOW_COMBINATORIAL_LOOPS : string;
attribute ALLOW_COMBINATORIAL_LOOPS of q_int : signal is "TRUE";
begin
    U2: y_LUT
        port map(
            Q => q_int,
            P=>P,
            En=>en,
            Y=>y_int
        );
    LUT_selfX : LUT6
        generic map (
            INIT => X"0000FFFFFFFFFFFFFF") -- Specify LUT Contents
        port map (

```

```

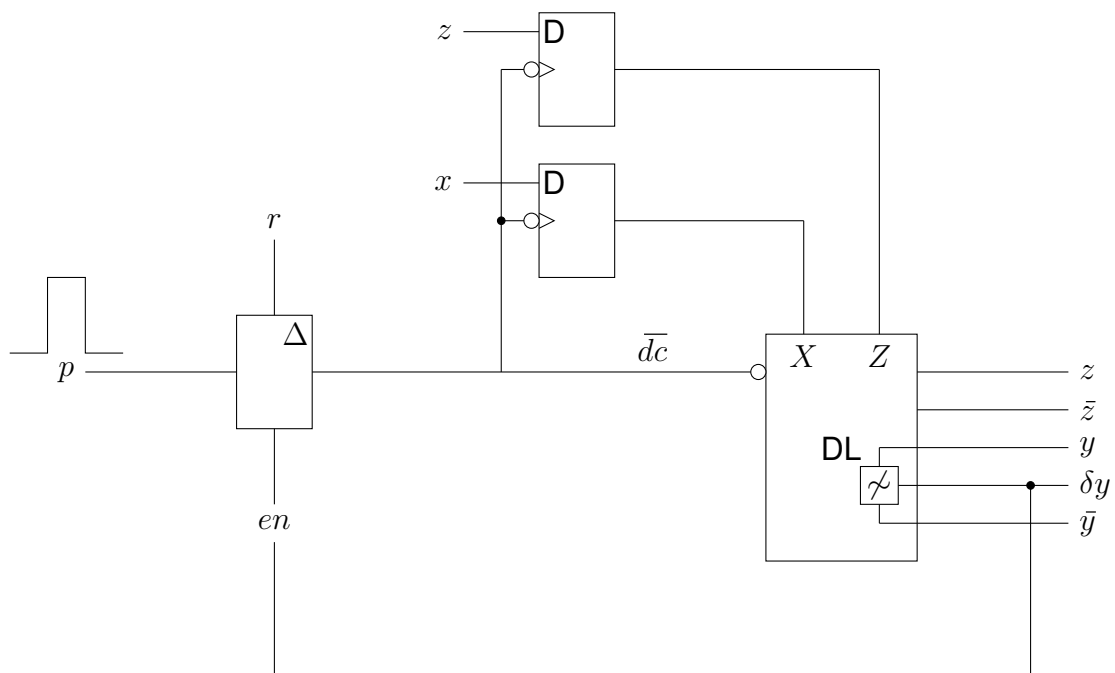
    0 => q_int, -- 6/5-LUT output (1-bit)
    I0 => q_int, -- LUT input (1-bit)
    I1 => '1', -- LUT input (1-bit)
    I2 => '1', -- LUT input (1-bit)
    I3 => '1', -- LUT input (1-bit)
    I4 => y_int, -- LUT input (1-bit)
    I5 => '1' -- LUT input (1-bit)
  );
dc<=q_int;

```

*Listing 4.2: Low-Level Selbstsperrende Pulsschaltung*

#### 4.2.4 Gesamte Dominologikschaltung

Die gesamte Schaltung auf GL ist in Abbildung 4.20 zu sehen.



*Abbildung 4.20: Selbstsperrende DRDL*

Da Platzierung und Routing grundlegend für die sichere Implementierung asynchroner Designs sind, ist es von höchster Bedeutung, die LUTs so anzuordnen, dass strukturelle Hazards oder funktional instabile Rückkopplungsschleifen ausgeschlossen werden.

In diesem Fall war es essenziell, die *LUT6* für die asynchrone selbstzurücksetzende Pulsschaltung zu verwenden, da dies das grundlegende logische Element des Artix-7

FPGA darstellt. Dadurch werden die Eingänge explizit definiert, was eine umfassende Nachvollziehbarkeit des Entwurfsprozesses ermöglicht.

Einige Aspekte des Designs erfordern jedoch weiterhin spezifische Constraints. Dazu gehören das Sperren der verwendeten Eingänge, das Zulassen kombinatorischer Schleifen sowie die Implementierung von do-not-touch-Befehlen (siehe Listing Listing 4.3), die eine Optimierung durch den Syntheseprozess verhindern, da dieser primär auf synchrone Entwürfe ausgerichtet ist.

```

component DRDL
port (
  dcbars: in std_logic;
  x: in std_logic_vector(3 downto 0);
  f_out: out std_logic;
  fbar_out: out std_logic
);
end component;
attribute dont_touch of DRDL : component is "yes";

```

*Listing 4.3: Constraints for Asynchronous FPGA Design*

#### 4.2.5 Serielle Dominologik-Pipeline mit Vollständigkeitserkennung

Es ist nun möglich, eine Reihe von Dominogattern so anzuordnen, dass eine Kaskade entsteht, die dann sequentiell betrieben wird. Dies steht im Gegensatz zur klassischen Pipeline-Verarbeitung, bei der mehrere Stufen gleichzeitig aktiv sind. Die Umsetzung erfolgt, indem für jede Übertragung einer 1 ein eindeutiger Zustand zugewiesen wird, also für jeden Domino-Effekt zur nächsten Stufe.

Allerdings ist die Verwendung einer echten Pipeline mit den erforderlichen Halteelementen zwischen den Stufen ebenfalls möglich. Die serielle Komposition wird durch die Verkettung der Dominogatter realisiert, beginnend bei der Eingangsstufe  $f_0$  bis zur Endstufe  $f_{n-1}$ , sodass die Funktion folgendermaßen definiert ist:

$$f = f_{n-1}(f_{n-2}(\dots(f_1(f_0))\dots)) = f_0 \circ f_1 \circ \dots \circ f_{n-2} \circ f_{n-1} \quad (4.11)$$

Zunächst werden alle Dominogatter aufgeladen, wodurch die internen Knoten auf eine Spannung von  $V_{DD}$  gebracht werden und die Ausgangswerte auf 0 gesetzt werden.

In der Evaluierungs-Phase werden dann alle DRDL-Gatter entlang des Signalpfades auf 0 gezogen, wodurch an genau einem der Ausgänge  $F$  oder  $\overline{F}$  eine 1 entsteht.

Sobald alle DRDL-Gatter in komplementären Zuständen sind, gilt das System als abgeschlossen, und der Eingang wird freigegeben.

Der Eingangsimpuls dient dabei als Request-Signal, während das *en*-Signal als Acknowledge fungiert, sodass ein asynchrones Handshaking-Protokoll realisiert wird.

In der Pipeline-Schaltung reicht es aus, lediglich die letzte Stufe auf Disjunktheit zu prüfen, da diese erst umschalten kann, wenn alle vorherigen Stufen bereits umgeschaltet haben. Dennoch wird in dieser Arbeit für jede Stufen die Disjunktheit ermittelt. Dazu werden die Ausgänge  $F$  und  $\bar{F}$  von jedem Dominogatter auf ein XOR-Gatter gegeben, welches 1 wird, wenn eine valide Signalkombination anliegt. Dies erhöht die Zuverlässigkeit und Sicherheit der Schaltung.

#### 4.2.6 Parallelisierung von Dominogattern

Das Umschalten von kombinatorischen Schaltungen kann in der Dominologik auch parallel erfolgen, anstatt in einer kaskadierten Reihenfolge. Da die selbstsperrende Dominologik gebündelte Daten annimmt und sich selbst taktet, wird ein Hazard-freier Betrieb auch in parallelisierten Dominogattern gewährleistet. Die Gatter können zu verschiedenen Zeiten fertig werden, da die Schaltung erst frei gibt, wenn alle Dominogatter geschaltet haben und in einem validen Ausgangszustand sind. Die Formel für die parallelisierte Dominologik lautet:

$f = f_0(x) + f_1(x) + \dots + f_{n-2}(x) + f_{n-1}(x)$ , wobei  $f_i(x)$  den Wert des  $i$ -ten Bit hat.

### 4.3 Vergleich der Entwurfsmethoden

Um die Entwürfe vergleichen zu können, wurde ein Automat gewählt, der total ist, d.h. er ist vollständig in den Zuständen - die Veroderung aller Zustände ergibt 1, als Gleichung:  $(\bigvee_{i=0}^{2^n-1} Z_i(x) = 1)$  - und vollständig in den ausgehenden Kanten pro Zustand - die Veroderung aller ausgehenden Kanten ist 1, siehe Gleichung 2.26. Der Automat ist zudem transient und oszillierend. Er ist nicht einschrittig, aber widerspruchsfrei und damit deterministisch - jede ausgehende Kante ist nur einmal pro Zustand vorhanden, siehe Gleichung 2.27. Der Automat ist eineindeutig bis auf Multiset und in Abbildung 4.21 als AG gegeben.

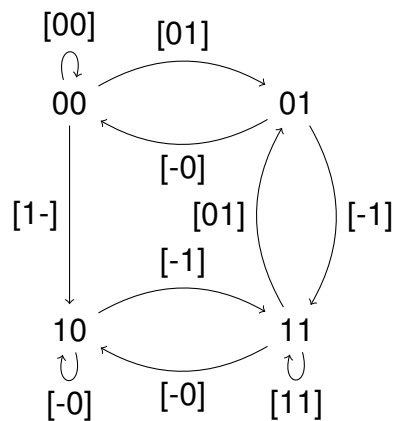


Abbildung 4.21: Beispielautomatengraph

### 4.3.1 Einschrittiger Entwurf

Der Automat ist einschrittig kodiert in den  $z$ -Variablen und muss damit nicht umkodiert werden. Da die Einschrittigkeit in den  $x$ -Variablen aber nicht vorliegt, werden die Eingänge so umkodiert, dass auch dieses Kriterium erfüllt ist, siehe Abbildung 4.22. Der Automat ist durch die Umkodierung partiell in manchen Eingangsbelegungen ge-

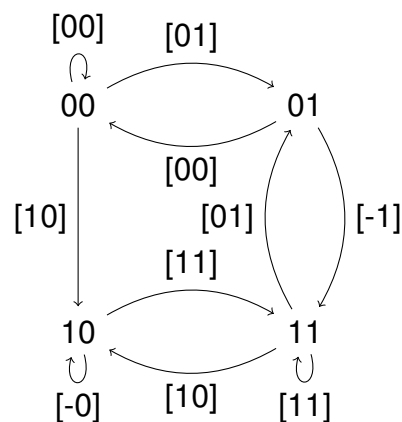


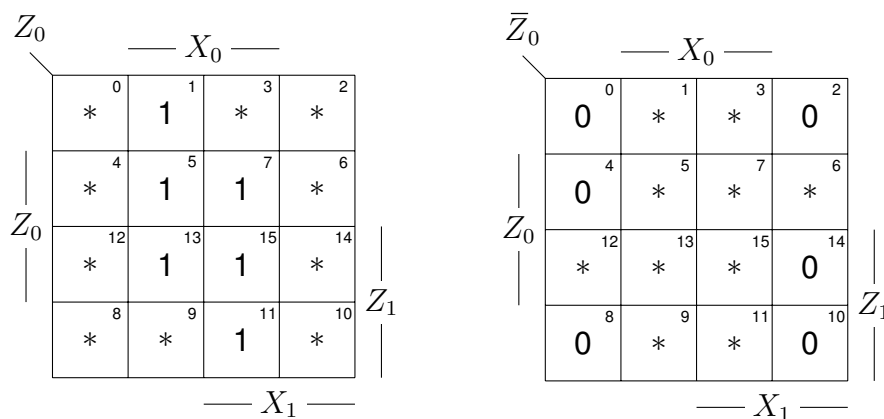
Abbildung 4.22: Umkodierter Beispielautomatengraph

worden, aber total in den Zuständen. Die partielle Zustandsüberführungstabelle ist in Tabelle 4.3 angegeben.

Als nächstes werden die einzelnen Zustandsvariablen  $z$  und  $\bar{z}$  in das KV-Diagramm eingetragen und die Resolvente berechnet, beispielsweise für  $z_1$  in Abbildung 4.23. Die undefinierten Kacheln liegen als \*-ne vor, d.h. diese Belegungen sind aus dem Kodierungsuniversum herausgenommen. Um einen totalen Automaten zu erhalten, werden die \*-ne als Hold kodiert, das bedeutet, dass der alte Zustand in die Kachel

Tabelle 4.3: Zustandsüberführungstabelle des partiellen Race-freien Automaten

$z_1$	$z_0$	$x_1$	$x_0$	$z_1$	$z_0$
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	*	*
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	*	*
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	*	*
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	*	*
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	1	1

Abbildung 4.23: Partielles KV-Diagramm von  $s_0$  und  $\bar{s}_0$ 

des KVD eingetragen wird, siehe Abbildung 4.24.

Die totalen Funktionen können nun im FPGA in den LUTs implementiert werden, dabei wird jedes Zustandspaar  $s_0$  und  $\bar{s}_0$  in einer Look-Up-Table LUT6\_2 realisiert. Dies ist möglich, da eine LUT6\_2, welche beide Ausgänge getrennt von einander bereitstellen muss, fünf Eingänge zur Verfügung hat und im Beispiel nur vier benötigt werden. Jedem Zustandspaar wird eine LUT für das funktionsstabile Muller-C-Element dazu geschaltet, um die Dual-Rail-Schaltungen zu vereinen. Die zugehörigen Formeln sind in Gleichung 4.12 bis Gleichung 4.18 angegeben, die resultierende Schaltung ist in

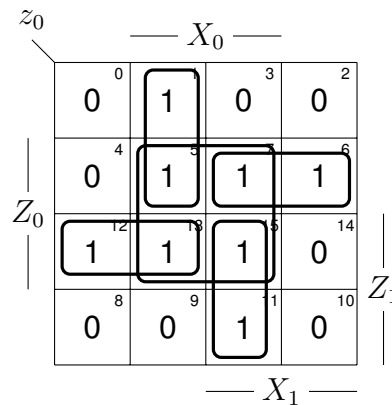


Abbildung 4.24: Totales KV-Diagramm von  $s_0 = (S_0, \bar{S}_0)$

Abbildung 4.25 zu sehen.

$$Z_0 := Z_0(X_0 \vee Z_1\bar{X}_1 \vee \bar{Z}_1X_1) \vee X_0(Z_1X_1 \vee \bar{Z}_1\bar{X}_1) \quad (4.12)$$

$$\bar{Z}_0 := \bar{Z}_0(\bar{X}_0 \vee Z_1\bar{X}_1 \vee \bar{Z}_1X_1) \vee \bar{X}_0(Z_1X_1 \vee \bar{Z}_1\bar{X}_1) \quad (4.13)$$

$$Z_0 := Z_0 + \neg(\bar{Z}_0) \quad (4.14)$$

$$Z_1 := Z_1(X_1 \vee \bar{X}_0 \vee \bar{Z}_0) \vee \bar{Z}_0X_1\bar{X}_0 \vee Z_0X_1X_0 \vee \bar{Z}_1Z_0X_0 \quad (4.15)$$

$$\bar{Z}_1 := \bar{Z}_1\bar{Z}_0(\bar{X}_1 \vee X_0) \vee \bar{Z}_1\bar{X}_1\bar{X}_0 \vee \bar{Z}_1Z_0\bar{X}_0 \vee Z_1Z_0\bar{X}_1X_0 \quad (4.16)$$

$$Z_1 := Z_1 + \neg(\bar{Z}_1) \quad (4.17)$$

$$(4.18)$$

### 4.3.2 Kaskaden-Entwurf

Im Folgenden wird die Eigenschaft der Krohn-Rhodes-Theorie genutzt, dass jeder Automat in eine kaskadierte Form gebracht werden kann, die aus Reset- und Permutationsgraphen besteht [49]. Es wird ein TSA aufgebaut [85], der zu einer kaskadierten Form führt. Der TSA wird aufgebaut, indem die gesamte Zustandsmenge gebildet wird, und die Zustandsübergangskantenmenge so aufgeteilt wird, dass Untermengen herauskommen, bis in der letzten Stufe nur noch einzelne Zustände übrig sind. Es wird zunächst mit der Inhibition von [11] auf der linken Seite und der Inhibition von [00] auf der rechten Seite der gesamten Teilmenge begonnen, siehe Abbildung 4.26. Alle ausgehenden Kanten außer [11] führen also in das Subset  $Z_0 \cdot Z_1 \cdot Z_3$ . Gleiches gilt für das Erreichen des Subsets  $Z_1 \cdot Z_2 \cdot Z_3$  durch Inhibition von [00] auf der rechten Seite. Nun sollen die Subsets noch kleiner werden, indem eine weitere Kanten herausgenommen

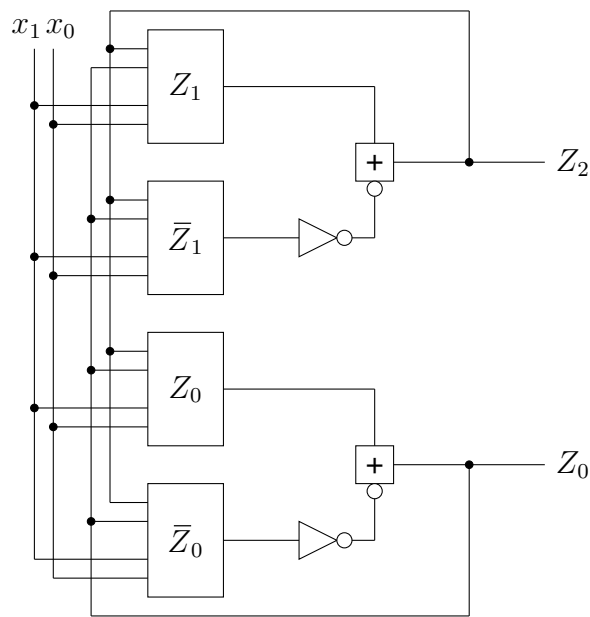


Abbildung 4.25: Einschrittige Dual-Rail-Implementierung der Beispielschaltung

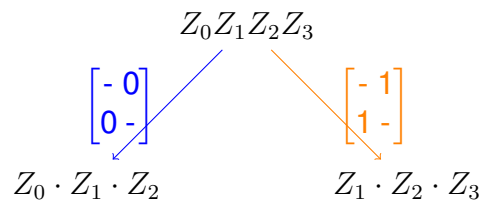


Abbildung 4.26: Entwicklung der ersten Stufe des TSA

wird, in diesem Fall auf der linken Seite die Kante [01], die Knoten der ersten Stufe werden also in  $[-0]$  und  $[-1]$  aufgeteilt, siehe Abbildung 4.27. Im letzten Schritt werden

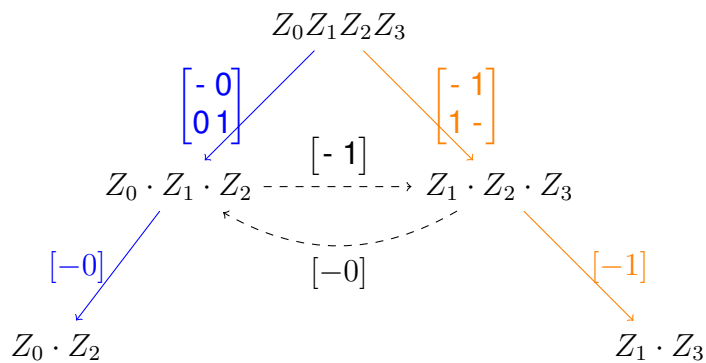


Abbildung 4.27: Teil-TSA mit Inhibitionen und Zustandsübergangskanten

die Zustandsüberführungskanten komplett disjunkt aufgespalten, sodass der TSA aus Abbildung 4.28 entsteht.

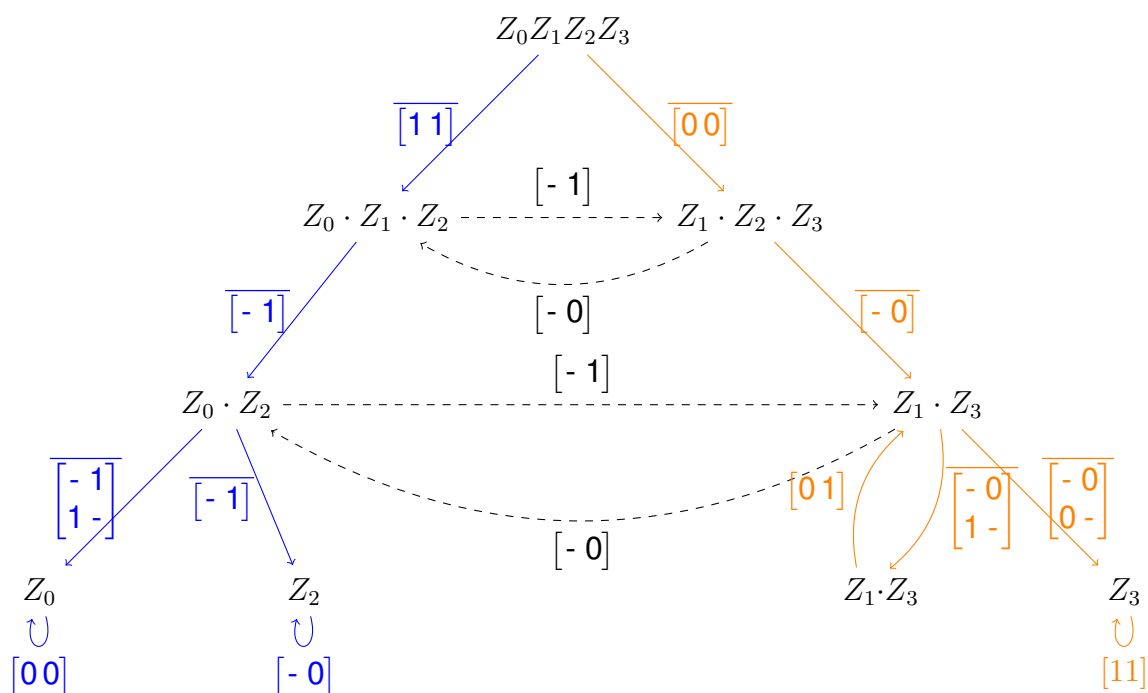


Abbildung 4.28: TSA mit Inhibitionen und Zustandsübergangskanten

Hier kann  $Z_1 \cdot Z_3$  nicht mehr verfeinert werden. Das liegt daran, dass eine Oszillation zwischen  $Z_1 \cdot Z_3$  vorliegt. Auch der Übergang von  $Z_0 \cdot Z_2$  über  $[00]$  ist nicht zu unterscheiden. Wenn für asynchrones Design mit funktionsstabilen eindeutig zugewiesenen Zuständen entworfen wird, kann der TSA-Algorithmus so wie er vorliegt angewendet werden, um Kaskaden aus funktionsstabilen Automaten zu realisieren.

Die Schaltung auf Gatterlevel ist in Abbildung 4.29 gegeben.

### 4.3.3 Vergleich der Entwürfe

Der Beispielautomat aus Abbildung 4.21 wurde sowohl als einschrittiger Automat als auch als Kaskade realisiert. In diesem Abschnitt sollen die beiden Entwurfsmethoden in ihrer Komplexität verglichen werden.

#### Einschrittiger Automatenentwurf

Der einschrittige Automatenentwurf erfolgt intuitiv: Zunächst werden die benötigten Zustände eines Automaten definiert, die dann so kodiert werden müssen, dass sie ausschließlich einschrittige Übergänge besitzen. Dies führt allerdings zu einer gewissen Redundanz, da teilweise transiente Zwischenzustände eingeführt werden müssen und



und damit die Zustandsdimensionierung exponentiell.

Dies bedeutet, dass die einschrittige Kodierung für große Automaten schnell unpraktikabel wird.

### **Kaskadierte Schaltung**

Das Aufstellen des TSA zur Erzeugung der Kaskaden-Struktur ist etwas aufwändiger, kann aber gut automatisiert werden.

Durch die Anwendung der TSA-Zerlegung wird die Zustandsmenge schrittweise in kleinere Teilmengen zerlegt. In jedem Zerlegungsschritt wird eine zusätzliche Zustandsdimension eingeführt, um die Trennung der Zustände zu gewährleisten. Im schlimmsten Fall ist die Anzahl der benötigten TSA-Stufen  $2^n - 1$ .

Die letzte Stufe kann dann wiederum eine Komplexität von  $2^n$  aufweisen, wodurch der maximale Aufwand  $2^n + (2^n - 1)$  beträgt.

In der Praxis lässt sich diese Komplexität reduzieren durch

- geschickte Zustandsreduktion, indem redundanten Zuständen des Ursprungsautomaten zusammengefasst werden oder Stufen des TSA, die das gleiche Ein- und Ausgangsverhalten haben, überlagert werden.
- Aufteilung des Ursprungsautomaten in einfachere Teilautomaten, sodass der TSA simple Automaten in eine kaskadierte Form bringt.
- Parallelisierung von nicht zusammenhängenden Teilautomaten.
- Änderung des Automatentyps. Mealy-Automaten sind gegenüber Moore- und Medwedew-Automaten zu präferieren, da sie i.A. weniger Zustände benötigen.

Dennoch bleibt die grundlegende Tendenz zu einem exponentiellen Wachstum bestehen.

## 5 Implementierung eines Asynchronen Multicycle Prozessors

In diesem Kapitel wird die Entwicklung eines Multicycle-Prozessors in einem FPGA vorgestellt. Zunächst erfolgt eine Einführung in den Mikroprozessorentwurf sowie eine Übersicht über die gesamte Prozessorarchitektur.

Darauf aufbauend wird die Control Unit als selbstsperrende Pipeline mit Dual-Rail-Dominologik entworfen und detailliert beschrieben. Anschließend erfolgt die Vorstellung der ALU sowie weiterer Komponenten des Prozessors. Die ALU wird ebenfalls asynchron gestaltet, um eine Handshaking-Kommunikation zwischen der Control-Unit und der ALU zu ermöglichen.

Im letzten Abschnitt wird die PPA-Analyse der FPGA-Implementierung durchgeführt, um die Effizienz und Machbarkeit des Systems zu bewerten. Die Ergebnisse belegen die Vorteile der asynchronen Implementierung hinsichtlich Leistungsaufnahme, Geschwindigkeit und Modularität.

### 5.1 Motivation für RISC-V

Über einen langen Zeitraum war der Markt für Prozessoren zwischen zwei Architekturen aufgeteilt: x86 und ARM, die hauptsächlich in mobilen Geräten eingesetzt werden. In den letzten Jahren ist jedoch mit RISC-V ein neuer Herausforderer aufgetreten, der einen neuartigen Ansatz verfolgt. RISC-V ist eine lizenzfreie Befehlssatzarchitektur (Instruction Set Architecture (ISA)), die an der University of California, Berkeley entwickelt wurde [86].

Im Gegensatz zur x86-Architektur, die sich über Jahre hinweg weiterentwickelt hat und durch eine hohe Komplexität gekennzeichnet ist, wurde die RISC-V-Architektur von Grund auf neu entwickelt, wobei das Prinzip der Einfachheit im Vordergrund stand. Das Ziel dieser Einfachheit ist zweigeteilt: Erstens soll die Hardware kostengünstiger werden, zweitens soll die Flexibilität erhöht werden. RISC-V wird zunehmend zu einem bedeutenden Akteur im Bereich der Prozessoren [87]. Ein entscheidender Vorteil besteht darin, dass es keinen Lizenzbeschränkungen unterliegt. Dies ermöglicht es einer Vielzahl von Unternehmen und Forschungsgruppen, Prozessoren auf Basis von RISC-V zu entwickeln und zu nutzen. Dies hat zur Entstehung einer breiten Palet-

te von RISC-V-Prozessoren geführt, die in verschiedensten Anwendungen eingesetzt werden. Das Spektrum reicht von Geräten mit hoher Energieeffizienz für das Internet der Dinge (IoT) bis hin zu Hochleistungsrechnern.

Die Einfachheit, Flexibilität und Lizenzfreiheit von RISC-V machen es zu einer attraktiven Option für viele Entwickler. Weitere positive Aspekte von RISC-V sind seine Energieeffizienz, Skalierbarkeit und Sicherheit, da die grundlegende Architektur so minimalistisch ist und nur wenig Angriffsfläche bietet.

## 5.2 GALS

GALS ist eine Entwurfsmethodik für elektronische Schaltungen. Sie adressiert die Herausforderung, eine sichere und zuverlässige Datenübertragung zwischen unabhängigen Taktdomänen innerhalb eines Systems zu gewährleisten.

Ein GALS-System unterteilt die Schaltung in unabhängige Blöcke, von denen jeder über einen eigenen lokalen Takt verfügt. Diese Blöcke kommunizieren asynchron über Handshaking-Protokolle [88]. Dies bietet mehrere Vorteile: Flexibilität, da die Blöcke mit unterschiedlichen Geschwindigkeiten arbeiten können, je nach ihren individuellen Anforderungen, sowie Skalierbarkeit, da das System ohne Rücksicht auf einen globalen Takt problemlos erweitert werden kann.

Darüber hinaus führt die GALS-Methodik zu einer reduzierten Leistungsaufnahme, da nur aktive Blöcke getaktet werden. Dies trägt maßgeblich zur Energieeffizienz des Systems bei.

## 5.3 Realisierung des RISC-V-Prozessors

Die Architektur von RISC-V lässt sich in zwei Hauptkomponenten unterteilen: den Datenpfad (Datapath) und die Steuereinheit (Control Unit). Diese Arbeit stellt das Design einer Steuereinheit für eine 32-Bit-Turing-vollständige RISC-V-Architektur vor.

In der RV32-RISC-V-Architektur sind sowohl Speicheradressen als auch Datenwörter 32 Bit breit. Dies stellt eine flexible und energieeffiziente Alternative zu den dominierenden RISC- und CISC-Architekturen dar. Der vereinfachte Befehlssatz von RISC-V ist klein und orthogonal, wodurch ein dynamisches Innovationsökosystem entsteht.

Dieser Ansatz reduziert die Hardware-Anforderungen und verbessert die Gesamtleistung, indem er die Komplexität und den Overhead herkömmlicher Befehlssätze elimi-

niert.

### 5.3.1 RISC-V Befehlssatz

Die grundlegenden Anweisungen mit einer Länge von 32 Bit werden anhand des Opcodes in verschiedene Kategorien unterteilt. Zur Veranschaulichung zeigt Tabelle 5.1 die Befehlssatzformate, die Turing-vollständig sind und in dieser Arbeit verwendet werden.

Tabelle 5.1: RISC-V Instruktionsformate

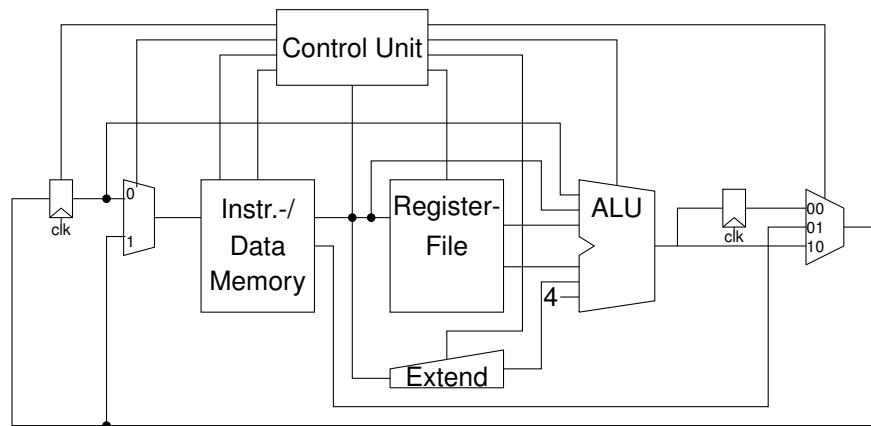
Type	Bits [31:25]	Bits [24:20]	Bits [19:15]	Bits [14:12]	Bits [11:7]	Opcode[6:0]
<b>R-Type</b>	funct7	rs2	rs1	funct3	rd	0110011
<b>I-Type</b>	imm[11:0]		rs1	funct3	rd	0010011
<b>L-Type</b>	imm[11:0]		rs1	funct3	rd	0000011
<b>S-Type</b>	imm[11:5]	rs2	rs1	funct3	imm[4:0]	0100011
<b>B-Type</b>	imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	1100011
<b>J-Type</b>	imm[20 10:1 11 19:12]				rd	1101111

Der Opcode wird in den ersten sieben Bits der Instruktion implementiert, während die verbleibenden Bits je nach spezifischer Operation variieren. Arithmetische Operationen verwenden überwiegend das R-Typ-Format. Eine solche Instruktion besitzt zwei Quellregister (rs1 und rs2) sowie ein Zielregister (rd) als Operanden. Die allgemeine Funktion der Instruktion wird durch das Funktionsfeld funct3 bestimmt, während die sieben Bits in funct7 verwendet werden, um die genaue Funktion zu spezifizieren, wenn zwei Instruktionen denselben funct3-Wert haben.

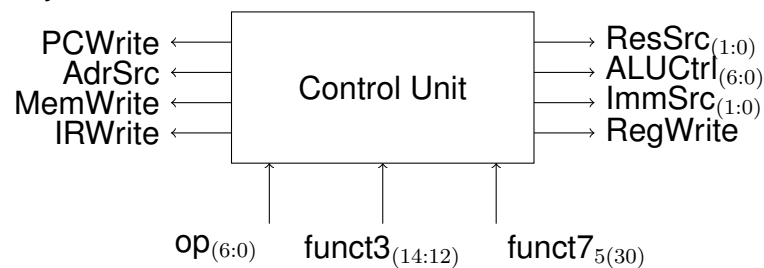
Ladeoperationen folgen dem L-Typ-Format, das identisch mit dem I-Typ-Format ist. Der Unterschied besteht darin, dass das L-Typ-Format für Speicherzugriffe verwendet wird. Das I-Typ-Format ist ähnlich dem R-Typ-Format, mit der Ausnahme, dass die Felder funct7 und rs2 fehlen. Stattdessen wird das 12-Bit-Feld imm genutzt.

Speicheroperationen verwenden das S-Typ-Format, bei dem der Wert aus rs2 an der durch imm und rs1 berechneten Adresse gespeichert wird. Das B-Typ-Format wird für Sprunganweisungen (Branching) genutzt, wobei die Werte in rs1 und rs2 verglichen werden und die Zieladresse in imm enthalten ist [89].

Die Sprung- und Verzweigungsanweisungen (Jump and Link) nutzen das J-Typ-Format, wobei die 20 Bits innerhalb des imm-Feldes die Zieladresse definieren, während die Rücksprungadresse in rd gespeichert wird [90].



(a) Architektur der Synchronen CPU



(b) Signalfussplan der Control Unit

Abbildung 5.1: Synchroner CPU und die synchrone Control Unit

### 5.3.2 Synchronous Multicycle CPU

Wir geben nun eine kurze Einführung in den ursprünglichen Prozessor [89], der in Abbildung 5.1 dargestellt ist. Es handelt sich um einen synchronen Prozessor, der Turing-vollständig ist, was bedeutet, dass er alle Turing-berechenbaren Funktionen ausführen kann.

Der Prozessor wird als Multi-Cycle-Prozessor realisiert, um unterschiedliche Zugriffszeiten für verschiedene Befehle zu ermöglichen. Dadurch kann eine Instruktion in einzelne, diskrete Verarbeitungsschritte unterteilt werden. Dies unterscheidet sich von einem Single-Cycle-Prozessor, bei dem der längste mögliche Pfad für die gesamte Instruktion berücksichtigt werden muss. Im Mehrzyklusansatz hingegen wird der längste Pfad pro Verarbeitungsschritt berücksichtigt, was eine effizientere Nutzung der Hardware ermöglicht.

Der Prozessor basiert auf einer Harvard-Architektur, was daran erkennbar ist, dass er separate Daten- und Instruktionsregister in einem Blockspeicher (BRAM) verwendet, das heißt, er besitzt zwei unterschiedliche Adressen für Instruktions- und Datenzugriffe.

### 5.3.3 Synchrone Steuereinheit

Zur Verarbeitung von Instruktionen wird ein Moore-Automat erstellt, der aus der RISC-V-Architektur abgeleitet wurde und den Opcode-Bereich 6:2 zur Dekodierung der einzelnen Zustände verwendet. Im Gegensatz zu [89] wurden einige zusätzliche Zustände integriert, da für den Speicherzugriff zwei Taktzyklen erforderlich sind.

Daher wird der Multi-Cycle-Prozessor in die folgenden Zweige unterteilt: Load, Store, R-Typ, I-Typ, B-Typ und JAL. Die Taktzyklen werden so verteilt, dass kürzere Zugriffszeiten erreicht werden, wie in Abbildung 5.2 dargestellt.

Dies führt dazu, dass die Ladeinstruktion die längste und die Branch-if-Equal (BEQ) Instruktion die kürzeste ist. Ein Single-Cycle-Prozessor müsste das ungünstigste Szenario für alle Instruktionen berücksichtigen, während der Multi-Cycle-Prozessor für die BEQ-Instruktion drei Taktzyklen einsparen kann.

### 5.3.4 Entwurf des asynchronen Controllers

Das asynchrone Handshake-Protokoll bietet das Potenzial, die Prozessorleistung erheblich zu verbessern, da es die Implementierung unterschiedlicher Zugriffszeiten für verschiedene Prozessschritte ermöglicht (z. B. ist das Schreiben in den Speicher deutlich langsamer als der Zugriff auf das Registerfile). Da die Dominanzlogik als Pipeline realisiert ist, ergibt sich eine direkte Übertragungsmöglichkeit auf Pipelines; dieser Aspekt wird jedoch in dieser Arbeit nicht weiter behandelt.

Um den Entwurf einer simpleren Kaskade zu ermöglichen, wird ein Mealy-Automat implementiert, der unterschiedliche Ausgabewerte für seine Zustände in Abhängigkeit vom Eingangssignal erzeugen kann. Aus dem synchronen Multi-Cycle-Moore-Automaten wird nun also ein Mealy-Automat, der manche Zustände in seiner Steuerung überlagert hat, die verschiedenen Zugriffszeiten durch Handshaking ausgleicht und verschieden lange Zyklen, je nach Eingang, generiert.

Im Gegensatz zum synchronen Moore-Automaten, der vordefinierte Verarbeitungszeiten für verschiedene Instruktionen besitzt, wird der Mealy-Automat extern durch REQ und ACK gesteuert. Dies bedeutet, dass identische Zustände für verschiedene In-

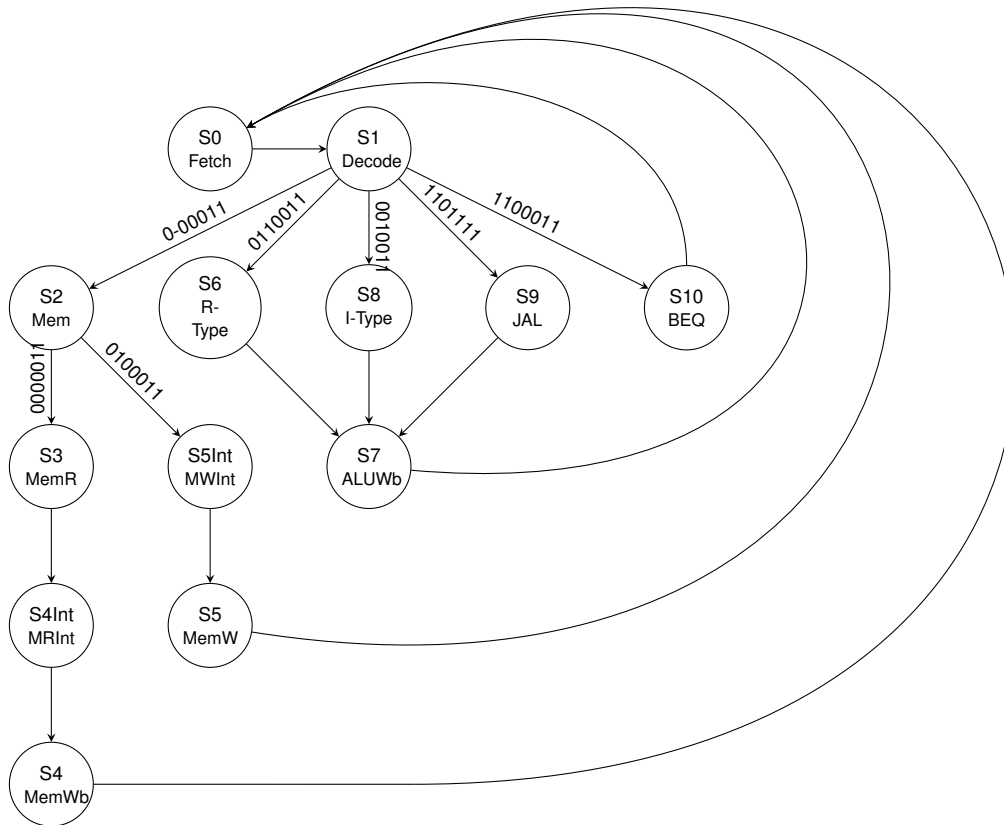


Abbildung 5.2: AG des synchronen Moore-Automaten

struktionen unterschiedliche Zugriffszeiten aufweisen können. Zudem wird die Selbstsperrung auch die Ausgabefunktion takten, wodurch Hazards und andere Probleme vermieden werden. Dies führt zu einer Reduzierung der Hardwareanforderungen im Vergleich zur Moore-Maschine. Die einzelnen Zustände wurden anschließend im One-Hot-Encoding kodiert.

Der AG der Schaltung ist in Abbildung 5.3 gegeben und seine Struktur in Abbildung 5.4.

Der Opcode des BEQ-Zweigs entspricht der Kante *A*, da er nur drei Zustandsübergänge erfordert. Kante *B* wird durch die R-Typ-, I-Typ- und JAL-Zweige definiert. Kante *C* repräsentiert den Übergang für die Store-Instruktion, während die Load-Instruktion den Endzustand erreicht, der durch die den Zustand [100000] dargestellt wird.

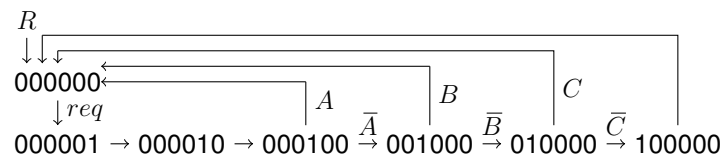


Abbildung 5.3: AG der Pipeline

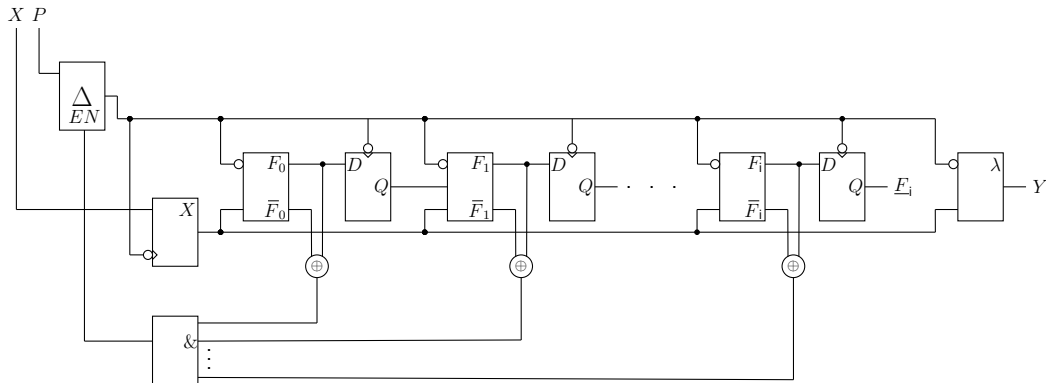


Abbildung 5.4: Struktur der realisierten Pipeline im FPGA

### 5.3.5 Entwurf der asynchronen ALU

Um die Vorteile des asynchronen Handshake-Protokolls effektiver zu veranschaulichen, wird die ALU als parallelisierte, selbstsperrende Dominologik realisiert. Dies ermöglicht eine umfassendere und genauere Darstellung der Funktionsweise des Handshaking-Protokolls. Angesichts der erheblichen Zeit, die für den Zugriff auf die ALU erforderlich ist, stellt die selbsttaktende Variante eine vielversprechende Verbesserung dar. Im Folgenden wird die 32 Bit umfassende AND-Funktion der ALU realisiert. Die resultierende Struktur des AND-Gatters für die ALU in selbstsperrender Dominologik ist in Abbildung 5.5 dargestellt. Die ALU hat natürlich auch am Eingang eine selbstsperrende Pulsschaltung, die einen Duty Cycle erzeugt und damit die Precharge-Phase einleitet. Anschließend erfolgt in der Evaluierungsphase das parallele Schalten jedes Gatters. Der Eingang wird erst freigegeben, wenn alle 32 DRDL-Gatter disjunkte Ausgänge haben.

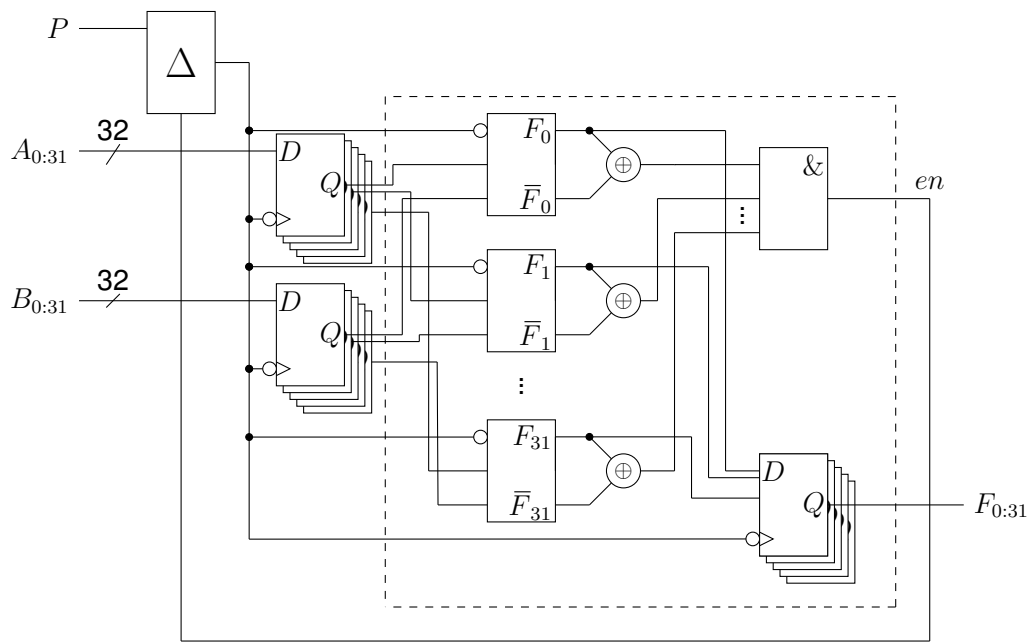


Abbildung 5.5: ALU aus parallelen Dominogattern

Der folgende Codeausschnitt zeigt die AND-Funktion der ALU als DRDL AND2.

```

MY_GEN : for i in 0 to 31 generate
  DominoGate: dualRail
  port map(
    dcbar => dc,
    x(0) => '1',
    x(1) => '1',
    x(2) => reg_b(i),
    x(3) => reg_a(i),
    f_out => f_int(i),
    fbar_out => fbar_int(i)
  );
  CompletionDetection: xor_LUT
  port map(
    A => f_int(i),
    B => fbar_int(i),
    Y => xor(i)
  );
  process (dc)
  begin
    if (falling_edge(dc)) then
      f_out(i) <= f_int(i);
    end if;
  end process;
end generate

```

```
        fbar_out(i) <= fbar_int(i);
    end if;
end process;
end generate;
process(xor)
begin
    if(xor=x"FFFFFFFF")
    then
        en_int<='1';
    else
        en_int<='0';
    end if;
end process;
```

*Listing 5.1: Low-Level 32-Bit DRDL AND*

### 5.3.6 Integration in die CPU

Der selbstsperrende Automat kann nun problemlos in die bestehende CPU integriert und beispielsweise über den Takt gesteuert werden, was die Einfachheit dessen Einbindung in bestehende Systeme zeigt. Zwar führt dies nicht unmittelbar zu einer Leistungssteigerung, sofern die anderen Prozessorkomponenten kein GALS-Verhalten aufweisen, doch verdeutlicht es die einfache Integration selbstgetakteter Schaltungen.

Darüber hinaus erfordert die Selbsttaktung weniger Flip-Flops, was zu einer reduzierten Leistungsaufnahme führt. Um die Vorteile des asynchronen Handshakes zu veranschaulichen, wurde die DRDL-ALU ebenfalls integriert. Die Steuereinheit verwaltet den Betrieb der anderen Komponenten auf synchrone Weise, während sie gleichzeitig den asynchronen Handshake-Prozess mit der ALU einleitet, wodurch die Ausführung von ALU-basierten Instruktionen beschleunigt wird.

## 5.4 PPA Ergebnisse

Die PPA-Analyse des implementierten asynchronen Prozessors auf einem FPGA zeigt eindeutig dessen Effizienz und Eignung für eine Vielzahl von Anwendungen.

Dieser Abschnitt präsentiert die Ergebnisse der Synthese, Implementierung und Simulation unter Verwendung der Vivado Design Suite. Die Resultate werden in drei Unterabschnitte gegliedert: Leistungsanalyse, Performanz-Analyse und Flächenanalyse.

### 5.4.1 Leistungsanalyse

Die Leistungsverbrauchsanteile der einzelnen Prozessoren wurden mit dem Vivado Power Analysis Tool ermittelt, wie in Abbildung 5.6 dargestellt.

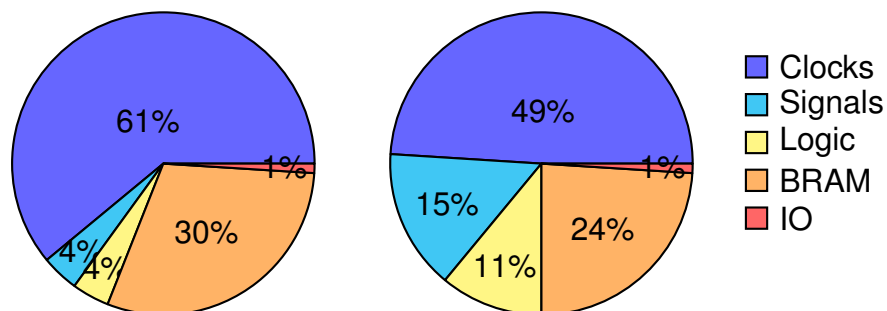


Abbildung 5.6: Leistungsverbrauch der Synchronen vs. Asynchronen CPU

Der gesamte dynamische Leistungsverbrauch sowohl des asynchronen als auch des synchronen Prozessors ist nahezu identisch.

Dennoch führt das asynchrone Design zu erheblichen Energieeinsparungen im Taktnetzwerk, wie sich an dem geringeren Anteil des Taktverbrauchs im Vergleich zum synchronen Prozessor zeigt.

### 5.4.2 Performanz-Analyse

In diesem Abschnitt wird die Performanz des asynchronen CPU-Designs mit der Performanz der synchronen CPU verglichen. Um die Vorteile sowie mögliche Kompromisse des asynchronen CPU-Designs im Vergleich zu seinem synchronen Gegenstück zu verdeutlichen, wird ein Instruktionsmix verwendet, der von der SPECint2000-Benchmark-Suite [91] inspiriert ist.

Die SPECint2000-Benchmark-Suite ist ein etabliertes und weit verbreitetes Werkzeug zur Leistungsbewertung von Prozessoren. Sie wird von der Standard Performance Evaluation Corporation (SPEC) entwickelt und gepflegt. Sie konzentriert sich speziell auf die Verarbeitung von Ganzzahlen und wird häufig zur Messung der Leistung von CPUs mit rechenintensiven Ganzzahl-Workloads verwendet.

Anstatt die vollständigen Programme der SPECint2000-Benchmarks auszuführen, wird in dieser Arbeit die charakteristische Verteilung der Instruktionstypen aus SPECint2000 verwendet. Der Instruktionsmix besteht zu etwa 25 % aus Ladeoperationen, 10 % aus Speicheroperationen, 11 % aus Sprunganweisungen, 2 % aus di-

rekten Sprüngen und 52 % aus R- oder I-Type-ALU-Instruktionen [89]. Diese Verteilung ermöglicht eine realitätsnahe, jedoch kontrollierbare Modellierung typischer CPU-Workloads.

Die synchrone CPU und die getestete asynchrone CPU wurden mit diesem definierten Instruktionsmix betrieben. Zur Durchführung der Messungen wurde ein Schleifenmechanismus implementiert, bei dem eine BEQ-Instruktion zur Steuerung des Programmflusses verwendet wurde. Die Ausführung erfolgte bei einer Taktfrequenz von 100 MHz. Anschließend wurden der Durchsatz der asynchronen und synchronen CPUs sowie die durchschnittliche Dauer pro Instruktion bestimmt, siehe Tabelle 5.2.

*Tabelle 5.2: Performanz-Kennzahlen*

Parameter	Asynchronous CPU	Synchronous CPU
Durchsatz (MIPS)	25.64	22.73
Durchschn. Latenz/Instruktion	39.5 ns	44 ns

Die Analyse zeigt, dass die CPU-Leistung für den Instruktionsmix durch die Nutzung von DRDL-Gattern zur Kommunikation zwischen der Steuereinheit und der ALU um etwa 10 % gesteigert werden konnte.

### 5.4.3 Flächenanalyse

Das Ziel der Flächenanalyse ist die Bewertung der Nutzung von FPGA-Ressourcen, einschließlich LUTs, Slice-Register und Slices. Das asynchrone Design nutzte 6,67% der verfügbaren LUTs, was auf eine moderate Komplexität in der logischen Implementierung hinweist. Darüber hinaus wurden 4,85% der verfügbaren Slice-Register und 4,63% der Slices verwendet.

*Tabelle 5.3: FPGA Ressourcenverwendung*

Resource Type	Utilization in (%) Async	Utilization in (%) Sync
LUTs	6.67%	6.32%
Slice Registers	4.85%	4.9%
Slices	9.27%	8.9%

Wie erwartet, nahm die genutzte Fläche zu, die Zunahme blieb jedoch im Rahmen, da die DRDL-Gatter innerhalb einer einzigen LUT implementiert wurden.

#### 5.4.4 Diskussion

Die PPA-Ergebnisse zeigen, dass der asynchrone Prozessor ein erhebliches Potenzial in Bezug auf die Performanz aufweist. Der Energieverbrauch zeigte nur leichte Verbesserung, dafür hielt sich der Flächenverbrauch in Grenzen. Die selbstsperrende Dominologikschaltungen zeigen, dass sie einfach in bestehende System zu integrieren sind und gleichzeitig zeigen sie Verbesserungen im Bereich der Performanz und Leistungsaufnahme.

### 5.5 Blueprint einer Asynchronen Multicycle CPU

Die einfache Integration, die sich aus der Modularität asynchroner Schaltungen ergibt, ermöglicht die Erstellung eines umfassenden Entwurfs für einen asynchronen Prozessor. Das Ziel besteht darin, den gesamten Prozessor asynchron zu gestalten, sodass jede Schaltungskomponente mit Request- und Acknowledge-Signalen arbeitet und somit kein globaler Takt für die gesamte Schaltung erforderlich ist, siehe Abbildung 5.7.

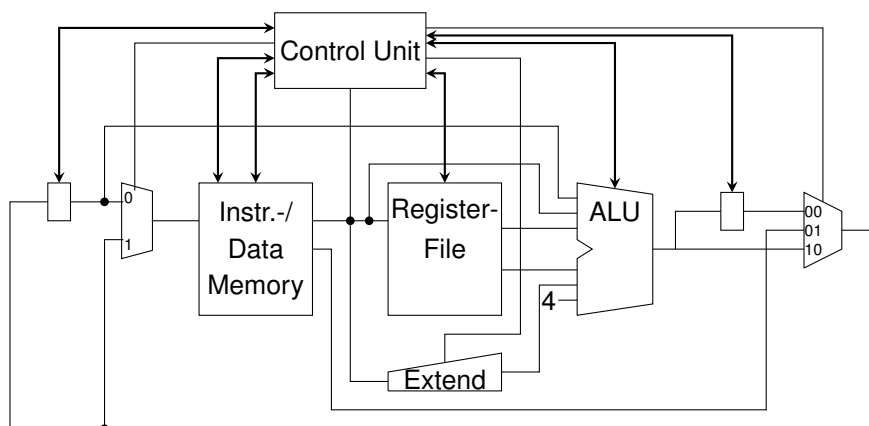


Abbildung 5.7: Asynchrone CPU

Es ist erkennbar, dass die einzelnen Schaltungskomponenten über ein Request- und Acknowledge-Mechanismus mit dem Controller kommunizieren, was durch das Doppelpfeil-Symbol dargestellt wird. Die Konstruktion kombinatorischer Schaltungen erfolgt unter Verwendung der selbstsperrenden Dominologik, während die Implementierung der DFFs durch asynchrone D-Latches mit Request- und Acknowledge-Signalen realisiert wird. Der Speicher arbeitet mit Taktzyklen, wobei das *ack*-Signal

---

präzise nach zwei Zyklen gesetzt wird, da der Datenzugriff diese Anzahl an Zyklen erfordert. Auch LUT-basierte Speicherstrukturen sind denkbar.

## 6 Zusammenfassung und Ausblick

### 6.1 Zusammenfassung

Diese Arbeit behandelt den asynchronen Entwurf von digitalen Schaltungen mit besonderem Fokus auf die Umsetzung in handelsüblichen FPGAs. Während synchrone Schaltungen den Designstandard in der Industrie darstellen, bieten asynchrone Systeme eine Reihe von Vorteilen, darunter einen geringeren Stromverbrauch, eine höhere Leistungsfähigkeit und die Eliminierung von Problemen, die durch globale Taktgeber entstehen. Allerdings sind bestehende FPGA-Architekturen in erster Linie auf synchrone Entwürfe optimiert, was eine besondere Herausforderung für den Einsatz asynchroner Methoden darstellt.

Zunächst werden die theoretischen Grundlagen erläutert, darunter die Logik und ihre verschiedenen Arten, die Automatentheorie, asynchrone Schaltungen, und relevante Logikdesign-Konzepte. Es werden verschiedene Automatentypen vorgestellt und ihre Einsatzmöglichkeiten analysiert. Zudem werden zentrale Problemstellungen wie Hazards, Races und Glitches betrachtet, die beim asynchronen Design eine große Rolle spielen.

Im weiteren Verlauf der Arbeit werden unterschiedliche FPGA-Hersteller und ihre Entwicklungsumgebungen betrachtet, mit besonderem Augenmerk auf die Freiheiten im Entwurf. Dabei werden Look-Up-Tables (LUTs), Speicherstrukturen und typische FPGA-Design-Flows betrachtet. Es wird gezeigt, wie sich asynchrone Schaltungen trotz der synchronen Natur von FPGAs effizient realisieren lassen. Besonders im Fokus steht die Implementierung eines funktionsstabilen Muller-C-Elements auf einer Artix-7 FPGA-Architektur sowie die Evaluierung der Performance und Stabilität solcher Designs.

Nach der Wahl des richtigen Herstellers werden Entwurfsmethoden für asynchrone Schaltungen untersucht, insbesondere: Einschrittiges Automatedesign, bei dem Zustände so kodiert werden, dass nur ein Bit pro Übergang wechselt, um Races zu vermeiden. Kaskadierte Entwürfe basierend auf der Krohn-Rhodes-Theorie, die allgemeine Automaten in kleinere, sequentiell verarbeitbare Teilautomaten zerlegen, um die Dominologik und damit stabile und fehlerresistente asynchrone Schaltungen aufzubauen.

Im abschließenden Teil der Arbeit wird die Realisierung eines asynchronen Multicycle-

Prozessors vorgestellt. Der synchrone Moore-Automat wird in einen asynchronen Mealy-Automaten umgewandelt und in einen bestehenden synchronen RISC-V Prozessor integriert. Um die Vorteile der GALS-Struktur mit ihrem Handshaking hervorzuheben, wird zudem die ALU als Vertreter einer kombinatorischen Schaltung als parallelisierte Dominologikschaltung aufgebaut. Abschließend wird der synchrone Prozessor mit seiner asynchronen Variante verglichen, wobei Leistungsaufnahme, Flächenbedarf und Verarbeitungsgeschwindigkeit (PPA-Analyse) untersucht werden. Die Ergebnisse zeigen, dass der asynchrone Prozessor Vorteile in der Performanz und Leistungsaufnahme bei überschaubarer Erhöhung der Fläche aufweist, während die Implementierung durch die Selbstsperrung und den Dual-Rail-Ansatz eine hohe Sicherheit gewährleistet.

## 6.2 Ausblick

Die Forschung auf dem Gebiet der asynchronen digitalen Schaltungen steht weiterhin vor großen Herausforderungen, insbesondere in der praktischen Umsetzung auf bestehende Hardwareplattformen. Während FPGAs eine flexible Umgebung bieten, ist ihr inhärent synchroner Charakter eine wesentliche Hürde für asynchrone Designansätze. Künftige Arbeiten könnten sich darauf konzentrieren, spezielle FPGA-Architekturen zu entwickeln, die asynchrone Entwürfe besser unterstützen, beispielsweise durch optimierte Routing-Algorithmen oder angepasste LUT-Architekturen. Hierfür können gezielt open-source Anwendungen verwendet werden, um bestehende Konzepte zu nutzen und anzupassen.

Zudem könnte die Integration von asynchronen Designmethoden in bestehende CAD-Tools die Entwurfsprozesse vereinfachen und automatisieren. Besonders vielversprechend ist die Anwendung asynchroner Systeme in energieeffizienten Embedded-Systemen, neuromorphen Chips und sicherheitskritischen Anwendungen, bei denen deterministische Laufzeiten von Vorteil sind.

Der in dieser Arbeit vorgestellte asynchrone Prozessor sollte als Pipeline realisiert werden, um noch effizienter zu sein. Hierfür eignen sich die asynchronen Handshaking-Protokolle und Selbstsperrung optimal. Denkbar sind dann auch LUT-basierte asynchrone Speicher, um Memoryzugriffszeiten zu minimieren.

Darüber hinaus wäre es sinnvoll, die in dieser Arbeit entwickelten Konzepte auf komplexere Systeme auszuweiten, um das Problem der Komplexität und deren Lösungen genauer zu untersuchen.

Zudem könnte die Verschmelzung von synchronen und asynchronen Entwurfsmethoden (GALS-Architekturen) ein vielversprechender Forschungszweig sein, um das Beste aus beiden Welten zu kombinieren.

Zusammenfassend liefert diese Arbeit eine grundlegende Methodik für den asynchronen FPGA-Entwurf und zeigt die Stärken sowie Herausforderungen dieses Ansatzes auf. Sie legt damit eine Basis für zukünftige Forschungen, die asynchrone Systeme noch effizienter und praktikabler für reale Anwendungen machen könnten.

## Literatur

- [1] F. Deeg, F. Eiermann und S. M. Sattler, "Verification of Function Stable Muller C-element in FPGA," in *AmE 2023 – Automotive meets Electronics; 14. GMM Symposium, 2023*, S. 62–67.
- [2] F. Deeg und S. M. Sattler, "Dynamic Effects in Asynchronous Circuits," in *AmE 2022 - Automotive meets Electronics; 13. GMM-Symposium, 2022*, S. 1–5.
- [3] F. Deeg, J. Zhu und S. M. Sattler, "Asynchronous Design," in *AmE 2020 - Automotive meets Electronics; 11th GMM-Symposium, 2020*, S. 1–5.
- [4] J. Zhu, "Zum Prototyping asynchroner Schaltungen mittels FPGA," Masterarbeit, Lehrstuhl für Zuverlässige Schaltungen und Systeme, Magisterarb., Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany, Feb. 2019.
- [5] J. An, "Realisierung Hazard-freier Asynchroner Schaltungen im FPGA," Masterarbeit, Lehrstuhl für Zuverlässige Schaltungen und Systeme, Magisterarb., Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany, Feb. 2022.
- [6] F. Eiermann, *Asynchrones CAD-Tool für FPGAs*, Bachelorarbeit, Lehrstuhl für Zuverlässige Schaltungen und Systeme, Erlangen, Germany, Sep. 2021.
- [7] F. Deeg, X. Wu und S. M. Sattler, "Self-locking Domino Logic Pipelined Controller for RISC-V in FPGA," *Athens Journal of Technology and Engineering*, Jg. 11, Nr. 3, S. 201–218, 2024. DOI: 10.30958/ajte.11-3-2.
- [8] F. Deeg und S. M. Sattler, "Self-Locked Asynchronous Controller for RISC-V Architecture on FPGA," in *AmEC 2024 - Automotive meets Electronics & Control; 15. GMM-GMA-Symposium, 2024*, S. 1–5.
- [9] F. Deeg, X. Chen und S. M. Sattler, "Self-Locking Domino Logic Pipelines: Application in RISC-V Architectures in FPGA," *Journal of Biosensors and Bioelectronics Research*, Jg. 2, Nr. 6, S. 1–10, 2024, ISSN: 2976-7466. DOI: 10.47363/JBBER/2024(2)126.
- [10] H. J. Zander, *Logischer Entwurf binaerer Systeme*. VEB Verlag Technik, 1989.
- [11] H.-D. Wuttke und K. Henke, *Schaltsysteme, eine automatenorientierte Einführung* (Informatik), ger. München: Pearson Studium, 2003, 352 S. ISBN: 3-8273-7035-3.

- [12] R. van Riel und G. Vosgerau, *Aussagen- Und Prädikatenlogik: Eine Einführung*. J.B. Metzler, 2018.
- [13] U. Kastens, *Skript zur Vorlesung Modellierung*, Uni Paderborn, 2009. Adresse: <http://www2.cs.uni-paderborn.de/cs/ag-klbue/de/courses/ws09/model/folien/kb-algebren.pdf>.
- [14] R. Der, *Skript zur Vorlesung Digitale Informationsverarbeitung*, Uni Leipzig, 2010. Adresse: <http://www.informatik.uni-leipzig.de/~der/Vorlesungen/DIV/logik.pdf>.
- [15] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, Jg. 72, Nr. 5, S. 593–599, 1953. DOI: 10.1109/TCE.1953.6371932.
- [16] W. Matthes, "Datenzugriffsprinzipien in objektorientierten Rechnerarchitekturen," Technische Universität Karl-Marx-Stadt (Chemnitz), Preprint, 1989.
- [17] W. Gehrke und M. Winzker, "Kombinatorische Schaltungen," in *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2022, S. 85–114, ISBN: 978-3-662-63954-2. DOI: 10.1007/978-3-662-63954-2\_4. Adresse: [https://doi.org/10.1007/978-3-662-63954-2\\_4](https://doi.org/10.1007/978-3-662-63954-2_4).
- [18] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, Jg. s2-42, Nr. 1, S. 230–265, 1937. DOI: <https://doi.org/10.1112/plms/s2-42.1.230>. eprint: <https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230>. Adresse: <https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230>.
- [19] M. Hofmann und M. Lange, *Automatentheorie und Logik*. Jan. 2011, ISBN: 978-3-642-18089-7. DOI: 10.1007/978-3-642-18090-3.
- [20] Y. T. Medvedev, "On the Class of Events Representable in a Finite Automaton," in *Sequential Machines: Selected Papers*, E. F. Moore, Hrsg., Addison-Wesley, 1964, S. 215–227.
- [21] A. Church, "Moore Edward F.. Gedanken-experiments on sequential machines. Automata studies, edited by Shannon C. E. and McCarthy J., Annals of Mathematics studies no. 34, litho-printed, Princeton University Press, Princeton 1956, pp. 129–153.," *The Journal of Symbolic Logic*, Jg. 23, S. 60, März 2014. DOI: 10.2307/2964500.

- [22] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, Jg. 34, Nr. 5, S. 1045–1079, 1955. DOI: 10.1002/j.1538-7305.1955.tb03788.x.
- [23] G. Uygur und S. M. Sattler, "A Real-World Model of Partially Defined Logic," in *12th International Workshop on Boolean Problems*, Freiberg, 2016.
- [24] K. Maheswaran, *Implementing Self-timed Circuits in Field Programmable Gate Arrays*. University of California, Davis, 1995. Adresse: <https://books.google.de/books?id=mNHTIWZPDmoC>.
- [25] S. Harris und D. Harris, *Digital Design and Computer Architecture: ARM Edition*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015, ISBN: 0128000562.
- [26] B. Steinbach und C. Posthoff, *Boolean Differential Calculus (Synthesis Lectures on Digital Circuits and Systems)*. Morgan & Claypool Publishers, 2017, ISBN: 9781627059220.
- [27] E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Switching Circuits," *IBM Journal of Research and Development*, Jg. 9, Nr. 2, S. 90–99, 1965. DOI: 10.1147/rd.92.0090.
- [28] S. H. Unger und S. Y. H. Su, "Asynchronous sequential switching circuits," 1969. Adresse: <https://api.semanticscholar.org/CorpusID:62749915>.
- [29] S. Unger, "Hazards, critical races, and metastability," *IEEE Transactions on Computers*, Jg. 44, Nr. 6, S. 754–768, 1995. DOI: 10.1109/12.391185.
- [30] D. A. Huffman, "The Design and Use of Hazard-Free Switching Networks," *J. ACM*, Jg. 4, Nr. 1, S. 47–62, Jan. 1957, ISSN: 0004-5411. DOI: 10.1145/320856.320866. Adresse: <https://doi.org/10.1145/320856.320866>.
- [31] S. M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers," Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1993. Adresse: <https://i.stanford.edu/pub/cstr/reports/csl/tr/95/686/CSL-TR-95-686.pdf>.
- [32] J. Beister, "A Unified Approach to Combinational Hazards," *IEEE Transactions on Computers*, Jg. C-23, Nr. 6, S. 566–575, Juni 1974.
- [33] H. Karthik und B. M. Kumar Naik, "Glitch elimination and optimization of dynamic power dissipation in combinational circuits," in *2014 International Conference on Advances in Electronics Computers and Communications*, 2014, S. 1–6. DOI: 10.1109/ICAIECC.2014.7002431.

- [34] R. Baumann und K. Kruckmeyer, *Radiation handbook for electronics*. Jan. 2019.
- [35] T. C. May und M. H. Woods, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE Transactions on Electron Devices*, Jg. ED-26, Nr. 1, S. 2–9, Jan. 1979.
- [36] J. Sparsø, "Asynchronous circuit design - A tutorial," 2001. Adresse: <https://api.semanticscholar.org/CorpusID:67482001>.
- [37] J. Sparsø, *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark, 2020. Adresse: <https://orbit.dtu.dk>.
- [38] G. J. Sayle, "Automated Synthesis of Delay-Insensitive Circuits," Ph.D. Thesis, University of Edinburgh, Edinburgh, UK, 1996. Adresse: <https://era.ed.ac.uk/bitstream/handle/1842/11374/Sayle1996.pdf>.
- [39] A. Bystrov, D. Shang, F. Xia und A. Yakovlev, "Self-Timed and Speed Independent Latch Circuits," *IEEE Transactions on Computers*, Aug. 1999, Source: CiteSeer. Adresse: [https://www.researchgate.net/publication/2357085\\_Self-Timed\\_and\\_Speed\\_Independent\\_Latch\\_Circuits](https://www.researchgate.net/publication/2357085_Self-Timed_and_Speed_Independent_Latch_Circuits).
- [40] T. Williams, "Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings," Ph.D. Thesis, Stanford University, Stanford, CA, USA, 1990. Adresse: <https://www-vlsi.stanford.edu>.
- [41] G. Uygur und S. M. Sattler, "Structure preserving modeling for safety critical systems," in *2015 IEEE 20th International Mixed-Signals Testing Workshop (IMSTW)*, 2015, S. 1–6. DOI: 10.1109/IMS3TW.2015.7177866.
- [42] G. Uygur, "Zur Parallelen Komposition von Automaten," Tag der mündlichen Prüfung: 18.12.2018, Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät, Erlangen, Deutschland, Dez. 2018. Adresse: <https://open.fau.de/items/95bb952f-3718-4e82-84c9-0582aee02bc4>.
- [43] D. Sokolov, J. Murphy, A. Bystrov und A. Yakovlev, "Design and analysis of dual-rail circuits for security applications," *IEEE Transactions on Computers*, Jg. 54, Nr. 4, S. 449–460, 2005. DOI: 10.1109/TC.2005.61.
- [44] D. E. Muller, "Theory of asynchronous circuits," English, University of Illinois at Urbana-Champaign. Graduate College. Digital Computer Laboratory, Urbana, Illinois, Report 66, Dez. 1955.

- [45] M. Özgül, F. Deeg und S. M. Sattler, "Mealy-to-Moore Transformation - A state stable design of automata," *Advances in Science, Technology and Engineering Systems Journal*, Jg. 2, Nr. 6, S. 162–174, 2017. DOI: 10.25046/aj020621.
- [46] M. Özgül, F. Deeg und S. M. Sattler, "Melay-to-Moore transformation in safety-critical systems," in *AmE 2017 - Automotive meets Electronics; 8th GMM-Symposium*, 2017, S. 1–5.
- [47] M. Özgül, F. Deeg und S. M. Sattler, "Mealy-to-moore transformation," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2017, S. 22–27.
- [48] D. Chapiro, "Globally-asynchronous locally-synchronous systems," Sep. 1984.
- [49] K. Krohn und J. Rhodes, "Algebraic Theory of Machines. I. Prime Decomposition Theorem for Finite Semigroups and Machines," *Transactions of the American Mathematical Society*, Jg. 116, S. 450–464, 1965, ISSN: 00029947, 10886850. Adresse: <http://www.jstor.org/stable/1994127> (besucht am 17.02.2025).
- [50] J. Rhodes und C. L. Nehaniv, *Applications of Automata Theory and Algebra: Via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy, and Games*. Singapore: World Scientific, 2009, ISBN: 978-981-283-696-3. Adresse: <https://www.worldscientific.com/worldscibooks/10.1142/6977>.
- [51] D. A. Hodges, H. G. Jackson und R. A. Saleh, *Analysis and Design of Digital Integrated Circuits*, 3rd. McGraw-Hill, 2004, ISBN: 978-0072283655.
- [52] S. D. Brown, R. J. Francis, J. Rose und Z. G. Vranesic, *Field-programmable gate arrays*. Springer Science & Business Media, 1992, Bd. 180.
- [53] S. M. Trimberger, *Field-programmable gate array technology*. Springer Science & Business Media, 2012.
- [54] S. Hauck und A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Burlington, MA, USA: Morgan Kaufmann, 2010, ISBN: 978-0-12-370522-8. Adresse: [https://books.google.com/books/about/Reconfigurable\\_Computing.html?id=dYKmZy0asrsC](https://books.google.com/books/about/Reconfigurable_Computing.html?id=dYKmZy0asrsC).
- [55] S. Roy, *Advanced Digital System Design: A Practical Guide to Verilog Based FPGA and ASIC Implementation*. Cham, Switzerland: Springer, 2023, ISBN: 978-3-031-41084-0. DOI: 10.1007/978-3-031-41085-7. Adresse: <https://link.springer.com/book/10.1007/978-3-031-41085-7>.

- [56] C. Chiasson und V. Betz, "Should FPGAs abandon the pass-gate?" In *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, S. 1–8. DOI: 10.1109/FPL.2013.6645511.
- [57] T. Pi und P. J. Crotty, "FPGA Lookup Table with Transmission Gate Structure for Reliable Low-Voltage Operation," US 6,667,635 B1, Primary Examiner: Don Le, Attorney, Agent, or Firm: Lois D. Cartier, 2003. Adresse: <https://patents.google.com/patent/US6667635B1/en>.
- [58] W. Wolf, *FPGA-Based System Design*. Upper Saddle River, NJ: Prentice Hall, 2004, ISBN: 0131424610.
- [59] M. Chirania und M. L. Vogel, "Lookup Table Circuits Programmable to Implement Flip-Flops," US 7,378,869 B1, Primary Examiner: Rexford Barnie, Assistant Examiner: Dylan White, Attorney: Lois D. Cartier, 2008. Adresse: <https://patents.google.com/patent/US7378869B1/en>.
- [60] *7 Series FPGAs Configurable Logic Block: User Guide*, Xilinx, 2016.
- [61] K. Tu, X. Xu, Y. Mei u. a., *FPGA EDA: Design Principles and Implementation*. Singapore: Springer, 2023, ISBN: 978-981-99-6577-4. DOI: 10.1007/978-981-99-6578-1. Adresse: <https://www.research-collection.ethz.ch/handle/20.500.11850/697890>.
- [62] M. Kubica, A. Opara und D. Kania, *Technology Mapping for LUT-Based FPGA*. Cham, Switzerland: Springer, 2021, ISBN: 978-3-030-60487-5. DOI: 10.1007/978-3-030-60488-2. Adresse: <https://link.springer.com/book/10.1007/978-3-030-60488-2>.
- [63] B. J. LaMeres, *Introduction to Logic Circuits & Logic Design with VHDL*. Cham, Switzerland: Springer International Publishing, 2017, ISBN: 978-3-319-34194-1. DOI: 10.1007/978-3-319-34195-8. Adresse: <https://www.springer.com/gp/book/9783319341941>.
- [64] R. Tönjes, *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. Wiesbaden, Deutschland: Springer Vieweg, 2020, ISBN: 978-3-658-29816-5. DOI: 10.1007/978-3-658-29817-2. Adresse: <https://link.springer.com/book/10.1007/978-3-658-29817-2>.
- [65] J. F. Wakerly, *Digital Design: Principles and Practices*, 3rd. Prentice Hall, 2000, Online version accessed March 2025. Adresse: [http://ebook.pldworld.com/\\_eBook/DIGITAL%20DESIGN%20PRINCIPLES%20&%20PRACTICES%203rd%20Edition/digital\\_design-third\\_edition-1.pdf](http://ebook.pldworld.com/_eBook/DIGITAL%20DESIGN%20PRINCIPLES%20&%20PRACTICES%203rd%20Edition/digital_design-third_edition-1.pdf).

- [66] R. Smith, "Design and Simulation of Aerospace Electric Thrusters Using FPGA," *Journal of Computer Technology and Software*, Jg. 1, Nr. 2, Apr. 2022. Adresse: <https://www.ashpress.org/index.php/jcts/article/view/45>.
- [67] J.-P. Deschamps, G. D. Sutter und E. Cantó, *Guide to FPGA implementation of arithmetic functions*. Springer Science & Business Media, 2012, Bd. 149.
- [68] H. F.-W. Sadrozinski und J. Wu, *Applications of field-programmable gate arrays in scientific research*. CRC Press, 2016.
- [69] M. Tehranipoor und C. Wang, *Introduction to Hardware Security and Trust*. Springer Publishing Company, Incorporated, 2011, ISBN: 1441980792.
- [70] Altera Corporation, *Altera Cyclone II Device Family Data Sheet*, Available from Altera Corporation, Juli 2005, S. 2–7.
- [71] Xilinx, *Artix-7 FPGAs Data Sheet: DC and Switching Characteristics*, [https://www.xilinx.com/support/documentation/data\\_sheets/ds181\\_Artix\\_7\\_Data\\_Sheet.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf), 2021.
- [72] Xilinx, "Power Reduction Techniques in FPGA Design: Clock Gating Strategies," *Xcell Journal*, Jg. 41, S. 24–29, 2000s. Adresse: <https://bitsavers.trailing-edge.com/components/xilinx/xcell/Xcell41.pdf>.
- [73] Xilinx, *Artix-7 FPGA Product Brief*, 2021. Adresse: <https://www.xilinx.com/support/documents/product-briefs/artix7-product-brief.pdf>.
- [74] *7 Series FPGAs Configurable Logic Block*. XILINX, UG474 (v1.8) September 27, 2016.
- [75] C. Pham-Quoc und A.-V. Dinh-Duc, "Hazard-free Muller Gates for Implementing Asynchronous Circuits on Xilinx FPGA," in *2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications*, 2010, S. 289–292. DOI: 10.1109/DELTA.2010.40.
- [76] *Vivado Design Suite 7 Series FPGA Libraries Guide*. XILINX, UG953 (v 2012.2) July 25, 2012.
- [77] *Vivado Design Suite User Guide*. XILINX, UG903 (v2020.1) August 17, 2020.
- [78] N. M. Morris, "Asynchronous Counters," in *Digital Electronic Circuits and Systems*. London: Macmillan Education UK, 1974, S. 105–109, ISBN: 978-1-349-01895-6. DOI: 10.1007/978-1-349-01895-6\_10. Adresse: [https://doi.org/10.1007/978-1-349-01895-6\\_10](https://doi.org/10.1007/978-1-349-01895-6_10).

- [79] S. Uniyal und V. Ramola, *A new 4 Bit Asynchronous Counter using Novel Low power explicit type pulse-triggered Delay Flip Flop (D-FF) 1*, Jan. 2019.
- [80] R. Parthier, *Messtechnik: Vom SI-Einheitensystem über Bewertung von Messergebnissen zu Anwendungen der elektrischen Messtechnik*. Springer Vieweg, 2020.
- [81] R. J. Nelson, "D. A. Huffman. The synthesis of sequential switching circuits. Journal of the Franklin Institute, vol. 257 (1954), pp. 161–190, 275–303.," *Journal of Symbolic Logic*, Jg. 20, Nr. 1, S. 69–70, 1955. DOI: 10.2307/2268078.
- [82] M. Elbably, "State Machine Transition to Avoid the Race Conditions in Asynchronous Sequential Logic Circuits," in *7th National Radioscience Conference (NR-SC)*, Egypt, Feb. 2000, S. 22–24.
- [83] J.-W. Kang, C.-L. Wey und P. Fisher, "Application of bipartite graphs for achieving race-free state assignments," *IEEE Transactions on Computers*, Jg. 44, Nr. 8, S. 1002–1011, 1995. DOI: 10.1109/12.403716.
- [84] T. Ndjountche, *Digital Electronics 3: Finite-state Machines*. Wiley-ISTE, 2016, ISBN: 978-1848219861. Adresse: <https://onlinelibrary.wiley.com/doi/book/10.1002/9781119371083>.
- [85] O. Maler, "On the Krohn-Rhodes Cascaded Decomposition Theorem," in *Time for Verification: Essays in Memory of Amir Pnueli*, Z. Manna und D. A. Peled, Hrsg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 260–278, ISBN: 978-3-642-13754-9. DOI: 10.1007/978-3-642-13754-9\_12. Adresse: [https://doi.org/10.1007/978-3-642-13754-9\\_12](https://doi.org/10.1007/978-3-642-13754-9_12).
- [86] A. Waterman, "Design of the RISC-V Instruction Set Architecture," 2016. Adresse: <https://api.semanticscholar.org/CorpusID:63861396>.
- [87] R.-V. International, "How the RISC-V ISA Offers Greater Design Freedom and Flexibility," März 2024, Accessed: 2024-10-20. Adresse: <https://riscv.org/news/2024/03/how-the-risc-v-isa-offers-greater-design-freedom-and-flexibility/>.
- [88] M. Krstic, E. Grass, F. K. Gürkaynak und P. Vivet, "Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook," *IEEE Design & Test of Computers*, Jg. 24, Nr. 5, S. 430–441, 2007. DOI: 10.1109/MDT.2007.164.
- [89] S. Harris und D. Harris, *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021, ISBN: 9780128200650. Adresse: <https://books.google.de/books?id=SksiEAAAQBAJ>.

- 
- [90] A. Waterman und K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2," 2017.
- [91] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, Jg. 33, Nr. 7, S. 28–35, Juli 2000, ISSN: 0018-9162. DOI: 10.1109/2.869367. Adresse: <https://doi.org/10.1109/2.869367>.

## A Quellcode und Messdaten

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.vcomponents.all;

library work;
use work.utils.ALL;
use work.rv32i_defs.ALL;

entity CTRL_AUTOMATON is
  Port ( X_4_IN : in STD_LOGIC;
        X_3_IN : in STD_LOGIC;
        X_2_IN : in STD_LOGIC;
        X_1_IN : in STD_LOGIC;
        dc_pint: out std_logic;
        CLK : in STD_LOGIC;
        RES : in STD_LOGIC;
        SET : in STD_LOGIC;
        instr  : IN std_logic_vector(xlen_range);
        F_PIN,F1_PIN,F2_PIN,F3_PIN,F4_PIN : out STD_LOGIC;
        FBAR : out std_logic_vector(4 downto 0);
        led: out STD_LOGIC;
        zero_flag: in std_logic;
        ADR_OUT : out STD_LOGIC := '0';    --addr out
        ALUA1_OUT : out STD_LOGIC; --alu src A : 00(PC) 01(old_pc) 10(
            rd1)
        ALUA0_OUT : out STD_LOGIC;
        ALUB1_OUT : out STD_LOGIC; --alu src B : 00(rd2) 01(imm) 10(4)
        ALUB0_OUT : out STD_LOGIC;
        OP1_OUT : out STD_LOGIC;    --aluop
        OP0_OUT : out STD_LOGIC;
        R1_OUT : out STD_LOGIC;    --result source
        R0_OUT : out STD_LOGIC;
        IRW_OUT : out STD_LOGIC;    --instruction write, instruction
            fetch eigentlich
```

```
    PCU_OUT : out STD_LOGIC;    --pc update
    REGW_OUT : out STD_LOGIC;  --register write
    MEMW_OUT : out STD_LOGIC;  --mem_write
    BRANCH_OUT : out STD_LOGIC; --BEQ
    alu_ctrl  : out alu_func;
    extension_unit_ctrl      : OUT extension_control_type;
    state_out : out std_logic_vector(3 downto 0);
    P : out std_logic;
    en : in std_logic;
    alu_out: OUT std_logic;
    rd1: OUT std_logic;
    rd2: OUT std_logic;
    data_reg: OUT std_logic
  );
  attribute dont_touch : string;
  attribute dont_touch of CTRL_AUTOMATON : entity is "true";
end CTRL_AUTOMATON;
```

```
architecture RTL of CTRL_AUTOMATON is
  signal LED_INT : STD_LOGIC_VECTOR(13 downto 0);
  signal fb : STD_LOGIC;
  signal leds : STD_LOGIC_VECTOR(13 downto 0);
  signal a,b,z: std_logic;
  signal p_alu: std_logic:='0';
  signal en_alu : std_logic:='0';

  component dualRail
  port (
    dcbar: in std_logic;
    x: in std_logic_vector(3 downto 0);
    f_out: out std_logic;
    fbar_out: out std_logic
  );
  end component;
  attribute dont_touch of dualRail : component is "yes";
```

```
component debouncerLUT is
port (
  P : in std_logic;
```

```
    dc : out std_logic;
    en : in std_logic
);
end component;
attribute dont_touch of debouncerLUT : component is "yes";

component completionDetection is
Port (
a: in std_logic;
b: in std_logic;
c: out std_logic
);
end component;
attribute dont_touch of completionDetection : component is "yes";

signal f : std_logic_vector(1 downto 0);
signal f_int : std_logic_vector(1 downto 0);

signal notf : std_logic_vector(1 downto 0);
signal notf_int : std_logic_vector(1 downto 0);
signal y : std_logic_vector(4 downto 0):="00000";
signal y_clkd: std_logic_vector(4 downto 0):="00000";

signal ybar: std_logic_vector(4 downto 0):="11111";
signal nP: std_logic:='1';
signal clocky: std_logic;
signal complete: std_logic_vector(4 downto 0);
signal en_int: std_logic;
signal start: std_logic;
signal request: std_logic:='1';
signal reset: std_logic:='0';
signal notReset: std_logic:='1';
signal x: std_logic_vector(3 downto 0);
signal ctrl_out: std_logic_vector(13 downto 0);
signal funct3_field : std_logic_vector(funct3_range);
signal alu_op_red : std_logic_vector(1 downto 0);
signal req_alu: std_logic;
signal asynch: std_logic;
signal rd2_int: std_logic:='0';

begin
```

```
u2detect: completionDetection
  port map(
    a=>y(0),
    b=>ybar(0),
    c =>complete(0)
  );

u2bdetect: completionDetection
  port map(
    a=>y(1),
    b=>ybar(1),
    c =>complete(1)
  );

u2cdetect: completionDetection
  port map(
    a=>y(2),
    b=>ybar(2),
    c =>complete(2)
  );

u2ddetect: completionDetection
  port map(
    a=>y(3),
    b=>ybar(3),
    c =>complete(3)
  );
u2edetect: completionDetection
  port map(
    a=>y(4),
    b=>ybar(4),
    c =>complete(4)
  );

  notreset<=not(reset);

U2: dualRail
  port map(
    dcbar => fb,
    x(0)=>'1',
```

```
x(1)=>'1',
x(2)=>notreset,
x(3)=>request,
f_out=>y(0),
fbar_out=>ybar(0)
);

--F_PIN<=y_clkd(0);
U2b:dualRail
port map(
dcbar => fb,
x(0)=>'1',
x(1)=>'1',
x(2)=>notreset,
x(3)=>y_clkd(0),
f_out=>y(1),
fbar_out=>ybar(1)
);
U2c:dualRail
port map(
dcbar => fb,
x(0)=>'1',
x(1)=>z,
x(2)=>notreset,
x(3)=>y_clkd(1),
f_out=>y(2),
fbar_out=>ybar(2)
);
U2d:dualRail
port map(
dcbar => fb,
x(0)=>'1',
x(1)=>a,
x(2)=>notreset,
x(3)=>y_clkd(2),
f_out=>y(3),
fbar_out=>ybar(3)
);
U2e:dualRail
port map(
dcbar => fb,
x(0)=>'1',
```

```
x(1)=>b,
x(2)=>notreset,
x(3)=>y_clkd(3),
f_out=>y(4),
fbar_out=>ybar(4)
);

process(X_4_IN,X_3_IN,X_2_IN,X_1_IN)
begin
if(X_4_IN='0' and X_2_IN='0' and X_1_IN='0') or (X_4_IN='1' and
    X_3_IN='1' and X_1_IN='0') then
a<='1';
else
a<='0';
end if;
end process;

process(X_4_IN,X_3_IN,X_2_IN,X_1_IN)
begin
if(X_4_IN='0' and x_3_IN='0' and X_2_IN='0' and X_1_IN='0') then
b<='1';
else
b<='0';
end if;
end process;

process(X_4_IN,X_3_IN,X_2_IN,X_1_IN,zero_flag)
begin
if(X_4_IN='1' and x_3_IN='1' and X_2_IN='0' and X_1_IN='0' and
    zero_flag='0') then
z<='0';
else
z<='1';
end if;
end process;

process(reset,fb)
begin
if(reset)='1' then
y_clkd <= "00000";
FBAR <= "11111";
end if;
```

```
    if (falling_edge(fb)) then
        y_clkd <= y;
        FBAR<=ybar;
    x(0)<=X_1_IN;
    x(1)<=X_2_IN;
    x(2)<=X_3_IN;
    x(3)<=X_4_IN;
        end if;
end process;

process(y_clkd)
begin
case(y_clkd) is
when "00000" =>

ctrl_out<="00010001011000";  --state 0

when "00001" =>

ctrl_out<="00101000000000";  --state 1

when "00010" =>
        case (x) is
when "0000" => --state 2

ctrl_out<="01001000000000";  --state 2

when "0100" =>

ctrl_out<="01001000000000";  --state 2

when "0110" =>

ctrl_out<="01000100000000";  --state 6
-- extension_unit_ctrl <= r_type;

when "0010" =>

ctrl_out<="01001100000000";  --state 8
```

```
extension_unit_ctrl <= i_type;

when "1101" =>

ctrl_out<="10101000001010"; --state 9
extension_unit_ctrl <= j_type;

when "1100" =>

ctrl_out<= "01000010001001";
extension_unit_ctrl <= b_type;

when others =>
end case;

when "00100" =>
    case (x) is
when "0000" =>

ctrl_out<="10000000000000"; --state 3

when "0100" =>

ctrl_out<="100000000000010"; --state 5

when "0110" =>

ctrl_out<="00000000000100"; --state 7

when "0010" =>

ctrl_out<="00000000000100"; --state 7

when "1101" =>

ctrl_out<="10110000001010"; --state 9  int

when "1100" =>

ctrl_out<="00101101001000"; --state add

when others =>
```

```
        end case;

when "01000" =>
    case (x) is
    when "0000" =>

        ctrl_out<="100000000000000"; --state 3int

    when "0100" =>

        ctrl_out<="0000000000000010"; --state 5int

    when "0110" =>

    when "0010" =>

    when "1101" =>

        ctrl_out<="000000000000100"; --state 7

    when "1100" =>

    when others =>
    end case;

when "10000" =>
    case (x) is
    when "0000" =>

        ctrl_out<="00000000100100"; --state 4

    when "0100" =>

    when "0110" =>

    when "0010" =>

    when "1101" =>

    when "1100" =>

    when others =>
    end case;
```

```
        when others =>

        end case;
    end process;

asynch<=clocky xor req_alu;

U3: debouncerLUT
    port map(
        P => asynch,
        en=> en_int,
        dc=>fb
    );

    process(complete, reset)
    begin
        if(reset ='1') then
            en_int<='1';
        elsif(complete ="11111") then
            en_int<='1';
        else
            en_int<='0';
        end if;
    end process;

    process(y_clkd)
    begin
        if(y_clkd = "00000") then
            request<='1';
        else
            request<='0';
        end if;
    end process;

    clocky<=CLK;

    state_out(0)<=y_clkd(0);
```

```
state_out(1) <= y_clkd(1);
state_out(2) <= y_clkd(2);
state_out(3) <= y_clkd(3);

funct3_field <= instr(14 downto 12);
alu_op_red(1) <= ctrl_out(8);
alu_op_red(0) <= ctrl_out(7);
process(alu_op_red)
begin

    case alu_op_red is
        when "00" => alu_ctrl <= func_add;
        when "01" => alu_ctrl <= func_sub;
        when "10" =>
            case funct3_field is
                when "000" => alu_ctrl <= func_add;
                when "001" => alu_ctrl <= func_sll;
                when "010" => alu_ctrl <= func_slts;
                when "011" => alu_ctrl <= func_sltu;
                when "100" => alu_ctrl <= func_xor;
                when "101" => alu_ctrl <= func_srl;
                when "110" => alu_ctrl <= func_or;
                when "111" => alu_ctrl <= func_and;
                when others =>
            end case;
        when others =>
    end case;
end process;
en_alu <= en;
process(x, en_alu, y_clkd)
begin
    case y_clkd is
        when "00010" =>
            P <= '1';
            case x is
                when "0010" =>
                    case en_alu is
                        when '1' =>
                            req_alu <= '1';
                        when others =>
                    end case;
            end case;
        end case;
    end case;
end process;
```

```
        end case;
    when "0110" =>
        case en_alu is
            when '1'=>
                req_alu<='1';
            when others=>
            end case;
        when others =>
        end case;
        when others=>
            P<='0';
        req_alu<='0';
    end case;
end process;

process(ctrl_out(9),ctrl_out(10))
begin
if ctrl_out(9)='0' and ctrl_out(10)='0' then
rd2_int <='1';
else
rd2_int <='0';
end if;
end process;

process(ctrl_out(6),ctrl_out(5))
begin
if ctrl_out(6)='0' and ctrl_out(5)='0' then
alu_out <='1';
else
alu_out <='0';
end if;
end process;

ADR_OUT    <= ctrl_out(13);
ALUA1_OUT  <= ctrl_out(12);
rd1        <= ctrl_out(12);
ALUA0_OUT  <= ctrl_out(11);
ALUB1_OUT  <= ctrl_out(10);
ALUB0_OUT  <= ctrl_out(9);
OP1_OUT    <= ctrl_out(8);
OP0_OUT    <= ctrl_out(7);
R1_OUT     <= ctrl_out(6);
R0_OUT     <= ctrl_out(5);
```

```
IRW_OUT    <= ctrl_out(4);
PCU_OUT    <= ctrl_out(3);
REGW_OUT   <= ctrl_out(2);
MEMW_OUT   <= ctrl_out(1);
BRANCH_OUT <= ctrl_out(0);

rd2<=rd2_int;
data_reg<=ctrl_out(7);

end RTL;
```

### *Listing A.1: Asynchrone Control-Unit*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
Library UNISIM;
use UNISIM.vcomponents.all;
use work.rv32i_defs.ALL;

entity ALU_AND is
Port ( P: in std_logic;
f_out_vec: out std_logic_vector(xlen-1 downto 0);
fbar_out_vec: out std_logic_vector(xlen-1 downto 0);
reg_a : in std_logic_vector(xlen-1 downto 0);
reg_b : in std_logic_vector(xlen-1 downto 0);
en : out std_logic
);
end ALU_AND;

architecture RTL of ALU_AND is

    component debouncer
    port (
    reset : in std_logic;
    P : in std_logic;
    dc : out std_logic;
    en : in std_logic
    );
    end component;
    attribute dont_touch : string;
    attribute dont_touch of debouncer : component is "yes";
```

```
    component dualRail2
    port (
        dcbar : in std_logic;
        x : in std_logic_vector(3 downto 0);
        f_out : out std_logic;
        fbar_out : out std_logic
    );
    end component;
    attribute dont_touch of dualRail2 : component is "yes";

    component xor_LUT
    port (
        A : in std_logic;
        B : in std_logic;
        Y : out std_logic
    );
    end component;
    attribute dont_touch of xor_LUT : component is "yes";

    signal lut_outputs_f : std_logic_vector(xlen-1 downto 0);
    signal lut_outputs_fbar : std_logic_vector(xlen-1 downto 0);
    signal xors : std_logic_vector(xlen-1 downto 0);
    signal dc : std_logic;
    signal en_int : std_logic:='0';

begin

    pulse: debouncer
    port map(
        reset =>'0',
        P => P,
        dc => dc,
        en => en_int
    );

    -- Instantiate 32 LUTs with unique names (LUT_0 to LUT_31)
    MY_GEN : for i in 0 to xlen-1 generate
    DominoGate: dualRail2
    port map(
        dcbar => dc,
        x(0)>=>'1',
        x(1)>=>'1',
```



Used LUT input:	A1	A2	A3	A4	A5	A6	Mean path delay of the feedback loop of the JKB [ps]:	Standard Deviation [ps]	0.269	0.284	0.194	0.253	0.137	0.072
A X0Y0	594,995	582,178	587,494	477,811	435,739	290,280	239,162	0,345	0,190	0,230	0,127	0,179	0,193	
B X0Y0	608,947	676,701	470,707	341,486	260,216	337,784	455,642	0,270	0,232	0,204	0,170	0,162	0,075	
C X0Y0	585,232	584,040	561,329	524,076	281,681	231,657	285,639	0,289	0,215	0,209	0,205	0,054	0,108	
D X0Y0	572,038	565,286	461,849	417,619	254,572	220,979	310,673	2,014	0,327	0,533	0,290	0,084	0,106	
A X1Y0	589,698	759,548	460,397	323,144	256,627	405,929	220,619	0,255	0,139	0,890	0,101	0,138	0,103	
B X1Y0	592,861	687,617	457,517	325,516	277,334	220,619	277,334	0,244	0,191	0,248	0,206	0,102	0,102	
C X1Y0	572,480	565,064	545,803	514,479	298,653	237,207	220,619	2,448	0,207	0,304	0,199	0,173	0,129	
D X1Y0	587,577	581,989	476,795	434,193	266,252	340,395	237,207	0,233	0,251	0,318	0,258	0,109	0,166	
A X2Y0	614,548	577,382	476,545	346,086	269,933	340,395	340,395	0,292	0,245	0,194	0,171	0,119	0,144	
B X2Y0	609,073	656,226	476,840	334,069	269,933	459,465	459,465	0,257	0,295	0,212	0,252	0,246	0,104	
C X2Y0	593,899	584,944	569,929	530,757	300,429	239,874	239,874	0,206	0,317	0,178	0,156	0,183	0,199	
D X2Y0	581,161	563,514	459,414	426,946	289,600	223,515	223,515	0,433	0,209	0,142	0,083	0,172	0,097	
A X3Y0	605,799	665,483	470,016	331,451	254,565	307,830	307,830	0,347	0,294	0,233	0,137	0,216	0,116	
B X3Y0	605,442	689,981	463,672	328,178	258,530	399,865	399,865	0,262	0,330	0,246	0,154	0,080	0,110	
C X3Y0	583,503	575,185	554,750	514,150	279,283	222,186	222,186	0,178	0,300	0,203	0,239	0,189	0,133	

Abbildung A.1: Messung der Feedback-Delays